软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.007277]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

Apache TsFile 中的短时间序列分组压缩及合并方法 *

刘星宇[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)

通讯作者: 宋韶旭, E-mail: sxsong@tsinghua.edu.cn

摘 要: 时间序列数据在工业制造、气象、电力、车辆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发展. 越来越多的数据库系统向云端迁移，端边云协同的架构也愈发常见，所需要处理的数据规模愈加庞大. 在端边云协同、海量序列等场景中，由于同步周期短、数据刷盘频繁等原因，会产生大量的短时间序列，给数据库系统带来新的挑战. 有效的数据管理与压缩方法能显著提高存储性能，使得数据库系统足以胜任存储海量序列的重任. Apache TsFile 是一个专为时序场景设计的列式存储文件格式，在 Apache IoTDB 等数据库管理系统中发挥重要作用. 本文阐述了 Apache TsFile 中应对大量短时间序列场景所使用的分组压缩及合并方法，特别是面向工业物联网等序列数量庞大的应用场景. 该分组压缩方法充分考虑了短时间序列场景中的数据特征，通过对设备分组的方法提高元数据利用率，降低文件索引大小，减少短时间序列并显著提高压缩效果. 经过真实世界数据集的验证，我们的分组方法在压缩效果、读取、写入、文件合并等多个方面均有显著提升，能更好地管理短时间序列场景下的 TsFile 文件.

关键词: 数据压缩；时间序列数据；数据库；工业物联网
中图法分类号: TP311

中文引用格式: 刘星宇,宋韶旭,黄向东,王建民. Apache TsFile 中的短时间序列分组压缩及合并方法. 软件学报. http://www.jos.org.cn/1000-9825/7277.htm

英文引用格式: Liu XY, Song SX, Huang XD, Wang JM. Short Time Series Group Compression and Merging Methods in Apache TsFile. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7277.htm

Short Time Series Group Compression and Merging Methods in Apache TsFile

LIU Xing-Yu[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, electricity, vehicles, and other fields, which has promoted the development of time series database management systems. More and more database systems are migrating to the cloud, and the architecture of end-cloud collaboration is becoming more common, leading to increasingly large data scales to be processed. In scenarios such as end-cloud collaboration and massive time series, a large number of short time series are generated due to short synchronization cycles, frequent data flushing, and other reasons, posing new challenges to database systems. Effective data management and compression methods can significantly improve storage performance, enabling database systems to handle the storage of massive time series. Apache TsFile is a columnar storage file format designed for time series scenarios, playing an important role in database

* 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)；国家电网公司总部科技项目(5700-202435261A-1-1-ZN)

收稿时间: 2024-05-27; 修改时间: 2024-07-16, 2024-08-19; 采用时间: 2024-08-29; jos 在线出版时间: 2024-09-13

management systems such as Apache IoTDB. This paper describes the group compression and merging methods used in Apache TsFile to address the scenario of a large number of short time series, especially in applications with a large number of time series such as industrial Internet of Things. This group compression method fully considers the data characteristics in the short time series scenario, improves the utilization of metadata through device grouping, reduces file index size, reduces short time series, and significantly improves compression efficiency. After validation with real-world datasets, our grouping method shows significant improvements in compression efficiency, reading, writing, file merging, and other aspects, enabling better management of TsFiles in scenarios with short time series.

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

时间序列数据是某些指标或物理量随时间变化而采集的数值所组成的数据序列，在工业制造、气象、电力、车辆等领域有着广泛应用. 在这些应用场景中，时间序列数据以其高频采集和连续性特点，形成了海量的数据流. 这些数据不仅需要实时处理，还需要高效存储，以满足实时监控和历史分析的需求.

端边云协同[1]是近年来数据库系统发展的一个重要趋势. 随着物联网设备的普及和云计算技术的成熟，越来越多的应用选择将数据采集与处理、存储与分析任务分布在本地终端设备和云端数据中心之间. 这种端边云协同模式能够充分利用终端设备的计算和存储能力，减轻云端服务器的负担，并提高数据处理的实时性.

海量序列场景[2]也越发常见. 随着物联网设备逐渐增多，各个应用场景中所要面对的数据规模越来越大，数据库管理系统需要处理的时间序列数量也越来越多，可达数亿级. 这类因为序列数量多而导致的难以管理、难以存储的问题称为海量序列难题或者高基数问题，在用户不断增长、存储序列不断增多的当下，海量序列的管理与存储成为一个具有巨大现实意义的问题.

在端边云协同场景中，端侧（terminal/device）、边侧（edge side）和云侧（cloud side）需要协作完成任务. 为了保证数据的实时性，端侧采集到的数据由边侧完成整合，需要定期同步到云侧，并且同步的周期可能很短. 此时，在短周期内采集到的数据量很少，会产生大量较短的时间序列，这种序列简称为短序列；对应的，较长的时间序列简称为长序列. 在海量序列场景中，也可能存在大量短序列. 这主要是由于数据库系统的每个分布式节点在短时间内需要处理的序列数量很多，会带来刷盘频繁、碎片文件较多等问题. 短序列会降低数据库系统的存储与管理效率，给云原生数据库系统带来新的挑战.

面对时间序列应用的巨大需求，以 Apache IoTDB[3]为代表的时序数据库，为管理时间序列数据提供了优秀的解决方案. Apache IoTDB 是一个面向工业物联网应用的时间序列数据库，集数据收集、存储和分析为一体，可以满足工业物联网等应用场景中海量存储、高速读取和复杂分析的需求[4]. 而 Apache TsFile[5]是 Apache IoTDB 的主要存储格式，它是一种专为时间序列数据设计的列式存储文件格式，支持高效压缩、高读写吞吐量，并且兼容多种框架[6].

面对短序列场景，现有的一些数据库文件格式面临的主要挑战可以总结为如下几个方面：

（1）时间序列的数据作为短序列被分散地存储在磁盘中，不仅导致大量元数据信息存在冗余，还拖慢了时间序列的查询效率.

（2）由于周期变化、重复模式等特征不明显，短序列的压缩效率一般较低.

（3）每条短序列都需要编制独立的索引信息，占用了高额的存储空间.

针对上述问题，我们提出了一种针对短时间序列场景特别设计的分组压缩方法，与 Apache TsFile 耦合. Apache TsFile 是 IoTDB 目前最主要的存储格式，使用列式存储（columnar storage）管理数据，并且将文件索引放在文件尾来加速查询. TsFile 以时间序列作为基本单位进行组织管理，将数据分页存储，这种不分组的管理模式对较长的时间序列表现较好；而在短序列场景下，绝大多数短序列的存储长度不超过一个数据页，且压缩效果差，这导致文件内有大量短数据页，空间利用率低. 通过我们的分组压缩方法改进后，TsFile 对短序列的存储效率有了显著提高. 在我们的分组压缩方法中，主要包括如下技术：通过对设备进行分组来得到设备组，针对设备组进行存储管理；利用设备组内序列的相似性，通过重复利用相同元数据信息的方法建立二级索引结构，利用两层索引降低索引存储开销；通过统一管理设备组内的序列，将短序列拼接为长序列，显著提高数据压缩效果.

本文的主要贡献有：

（1）根据真实应用场景，分析短序列应用场景的特点，提出针对该问题的分组压缩方法；

（2）设计并实现了一种与 Apache TsFile 耦合的分组压缩方法，将有相同特征的短时间序列集中存储，显著提高了 Apache TsFile 针对短序列场景下的存储表现；

（3）利用短时间序列片段的相似性，将同物理量名称的时间序列拼接存储，达到降低数据压缩比的目标；

（4）利用短序列场景下元数据的相似性，将 TsFile 原有的一层索引结构扩展为两层索引，减少了冗余元数据的存储，减少了文件索引的存储开销.

本文在第 1 节介绍数据库领域数据存储、数据索引、短序列压缩等方面的相关工作. 第 2 节主要介绍理论概念和基础知识. 第 3 节详细阐释本文提出的 Apache TsFile 分组压缩方法. 第 4 节详细介绍大量存在短时间序列的场景，并分析了本文提出的分组压缩方法在这些场景中的应用. 第 5 节通过实验来验证所提方法的有效性和主要优势. 最后在第 6 节总结全文.

1 相关工作

短时间序列场景给数据库系统带来了数据存储低效、查询速度慢等困难，本文主要研究的方向在于解决存储低效的问题，该问题又可以分为两个方向：其一是优化元数据信息与索引的存储，其二是通过压缩提高数据的存储效率.

1.1 数据库系统及其索引

如何在文件中构建数据表的索引一直是数据库管理系统需要解决的重要问题，索引的结构如何设计也至关重要. 一般来说，利用索引可以快速定位数据在文件中的位置，从而高效的进行数据读写. 而在短序列场景下，随着数据量与时间序列规模提升，索引的数量也随之增加，索引的存储负担也不断增大.

Apache TsFile 将各条时间序列的索引放在文件尾部，这样可以快速读取各序列的索引信息，从而快速定位数据块的位置；索引可以抽象地看作一个二元组，分别存储着时间序列的名称和代表文件数据块位置的整数指针. 随着时间序列内数据点数量的增加，单一的索引不能快速定位到需要查询的数据，此时记录数据位置的指针也会增多. 但是面对海量序列场景下时间序列数量庞大的情况，索引数量会随着序列数量线性增加，导致索引的存储开销快速增加，在我们调研的真实场景中索引的存储占比甚至可能高达 70%. 而在本文中，我们提出了一种与 TsFile 文件格式兼容的分组压缩方法，它能通过分组后再构建二级索引的方法降低索引的存储负担，同时保证索引的效率.

InfluxDB[7]是一个流行的开源时序数据库，使用时间结构的合并树（Time Structured Merge Tree）作为存储引擎[8]，并不断更新迭代. InfluxDB 的时间线索引系统（TSI，Time Series Index）是一种多层次的索引策略. TSI 的核心思想是将时间序列数据按时间段划分为块（blocks），每个块包含一组时间序列数据点,并对应一个索引文件，用于快速查找和访问其中的数据. 然而，和 TsFile 一样，在这样的索引模式下，索引大小会随着序列数量线性增加，在海量序列场景下面临索引大小膨胀、存储与查询性能下降等问题.

为了解决索引问题，InfluxDB 提出了 InfluxDB IOx[9]. 在 InfluxDB IOx 中，也采用了列式存储，与传统的行式存储相比，列式存储具有更好的压缩性能和查询性能. 同时，InfluxDB IOx 还引入了基于数据分区（data partitioning）的存储和查询策略，将数据按照时间范围和其他维度进行划分，以提高查询性能和降低索引大小. 此外，InfluxDB IOx 还采用了更灵活和可扩展的索引结构，能够更好地处理大规模时间序列数据的存储和查询需求.

Apache HoraeDB[10]是一款高性能、分布式的云原生时序数据库. HoraeDB 的基础设计思想是采用混合存储格式和相应的查询方法，以便在处理时序和分析场景时都获得更好的性能. 在经典时序数据库中，标签列（被称为 Tag 或 Label）通常使用倒排来进行索引，这样可以根据标签快速定位到时间序列. 然而在某些场景下，标签的基数会非常大. 在设计之初，HoraeDB 想要解决的主要问题是标签列基数非常高时编制索引开销

太高. HoraeDB 使用了分析型数据库中扫描与剪枝的思想，加速数据的索引.

1.2 数据的编码与压缩

在时间序列存储的过程中，编码与压缩方法能够显著降低存储负担，因此在存储时间序列信息时往往会进行各种形式的编码与压缩. 肖今朝[11]、夏天睿[12]等人分析了 Apache TsFile 中各类编码、压缩方法的效果，并且分析了在时间序列呈现不同时序特征时编码效果的变化，但是并没有分析短时间序列对编码效果的影响. 当一段数据被分散存储在多个短序列时，往往编码方法效果会显著下降，主要有两方面原因：（1）编码方法一般利用了序列中的周期变化、重复模式特征，短序列的特征会更不明显；（2）每个短序列都需要分别存储编码方法提取出的特征，这引入了额外开销. 因此，短时间序列的整体编码、压缩效果一般会下降，压缩比更高. TsFile 支持多种编码方法[12]，例如游程编码[13]（RLE）、差分编码、SPRINTZ 编码[14]、字典编码[15]、Gorilla 编码[16]、CHIMP 编码[17]等，这些编码方法都会受此影响. 以最易于理解的字典编码为例，当若干数据点被存储在多个短序列时，每个短序列都需要存储一个字典信息，并且不能重复利用字典中相同的键值对，因此效率会明显下降. 而在本文中，我们通过将分组内具有同样特征的时间序列做拼接处理，使得压缩方法所处理的数据点足够多，在不改变编码方法本身的情况下优化了总体的压缩比.

此外，也有相关研究提出了将时间序列分组来实现高效存储. 方晨光[18]等人提出了一种启发式算法，分析不同时间序列数据点分布的相似性，将相似的序列集中存储，以提高存储效率. 该方法更关注特定序列的时间戳相似性，能重复利用时间戳来降低存储负担. 此外，方晨光等人还研究了一种多列压缩方法[19]，旨在解决负载密集的场景中存储引擎空间放大（space amplification）的问题. 而本文的分组压缩方法更关注时间序列类型、名称等元数据的相似性，提取短序列场景下时间序列的共有特征，将有共同特征的时间序列集中存储，减少冗余元数据、冗余索引；通过短序列拼接的方法降低数据压缩比，提高存储效率.

2 基础知识

2.1 TsFile的数据存储模型

在 Apache TsFile 中，数据主要以树形结构进行组织和管理[5]. 如图 1（a）所示，数据在 Apache TsFile 文件中呈现树状结构. 从树根表示的 root 节点开始，自上而下每层分别是根目录、若干个数据库、中间结点层、设备层、物理量层. 根据用户的需要，中间结点可以没有，也可以存在多个. 从根节点到叶子节点的唯一路径，就唯一地表示一条时间序列. 例如路径 root.database1.wf1.device1.temperature，就表示了一条时间序列，它所属的数据库是 root.database1，所属的设备为 root.database1.wf1.device1，物理量名称为 temperature，这条时间序列可能存储了某个风电厂的某个设备所采集到的温度数据. 用户可以根据具体的使用场景，自定义树状结构的模型，对采集到的时间序列数据进行管理[4].

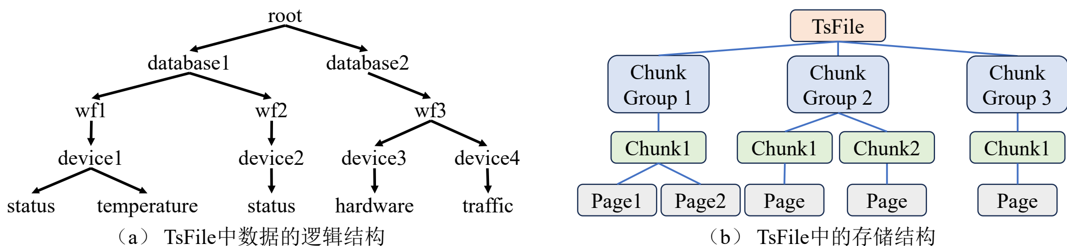


图 1 TsFile 中时间序列的存储逻辑和结构

Apache TsFile 的磁盘存储结构与其数据的逻辑结构逻辑遥相呼应. 图 1（b）展示了 TsFile 中数据的存储结构，一个 TsFile 包含了多个数据块存储组（Chunk Group），每个数据块存储组负责存储一个设备的信息，包含若干个数据块（Chunk）. 每个数据块则存储一条时间序列的数据，由元数据信息和若干个数据页（Page）

组成. 每个数据页中的时间序列采用列式存储，包括两列：一列是时间戳列，一列是数值列，两列分开编码、压缩、存储，因此是隔离的. 数据页是 TsFile 数据读写的基本单元，一般情况下需要完整读取整个页，才能解析其中存储的数据. 但是数据页、数据块、数据块存储组都含有数据摘要，描述了所存储数据的特征，例如数据点的最大值与最小值、起始时间与结束时间等信息. 这样，在查询过程中，可以利用数据摘要进行过滤，跳过无需读取的数据页.

2.2 数据编码与压缩

数据编码关乎数据存储的效率与质量，是在保证数据完整的前提下，减小存储空间占用的方法. 在数据存储的过程中，需要对数据进行编码，从人们易读的形式转化为计算机易于归类、排序、压缩的形式. 也即使用一定的算法，将数据序列化为二进制字节流，而转换后的二进制字节流通常比原有的数据表示占用更少的空间，以此来取得压缩效果. 数据压缩是通过减少数据冗余以节省存储空间或传输带宽的过程，根据是否丢失信息可以分为有损压缩和无损压缩.

在 Apache TsFile 文件的编码压缩方法中，时间序列的时间戳列与数值列是单独分开存储的. 这样分开的存储逻辑有诸多优势，例如可以针对数值列的类型采用有针对性的编码压缩方法，并且不同序列之间隔离且互不干扰. 此外，将时间戳列分离出来，也就能对时间戳进行有针对性的编码压缩，例如采用二阶差分编码来编码时间戳，进一步提高效率. 然而，这种分离存储的策略在大量短序列场景下面临挑战. 由于序列是分开存储的，因此即使它们有类似的元数据，比如相同的类型、编码方法、以及实际物理含义，也需要单独存储元数据，在序列数量极多的场景下带来了大量开销.

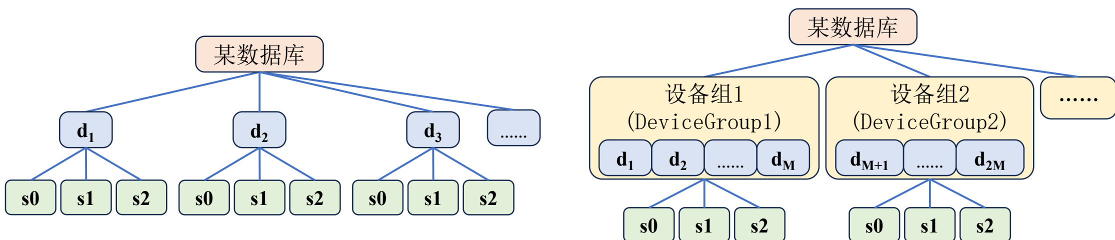
3 基于 Apache TsFile 的分组压缩方法

我们通过分析企业合作伙伴提供的真实世界数据集，发现短序列场景具有一些特征. 在这些场景中，时间序列的元数据信息非常规律且相似，这是由真实世界的特征所决定的. 一般情况下，应用场景中的设备一般是统一化、流水化的，因此有大量设备负责相同的职能，产生了相同类型的时间序列数据，例如气象采集设备监控的温度湿度数值、风电厂的电机转动速率、车辆行驶过程中发动机的转速等. 相同职能的设备采集的时间序列可能有相同或相似的特征，例如名称、物理量类型、数值范围等. 因此，很多短序列共用了相同的元数据信息，但在存储时被隔离开了，这造成了空间的浪费.

我们针对短序列场景的这一特征，提出一种分组压缩方法，能够充分利用重复的元数据信息，提高数据存储的效率. 接下来，我们将通过模型设计、压缩方法的设计、索引结构的设计等方面介绍分组压缩方法.

3.1 模型与分组方法

分组压缩方法沿用 TsFile 中的树模型进行建模，但是修改了设备的组织方式，我们在图 2 中列出了不分组方法和分组压缩方法的存储结构示意图. 通过将多个设备的信息整合到同一个节点，可以减少树形结构的节点数量，以此降低存储负担；同时，树形结构的每个叶子节点代表了一个磁盘上的序列存储单位，随着树形结构被剪枝，序列存储更有条理，减少了索引的存储负担.



（a） 不分组的TsFile存储结构 （b） 分组压缩的TsFile存储结构

图 2　TsFile 中的设备组模型

如图 2（a）所示，在不分组的 TsFile 树形存储模型中，存在若干设备，每个设备的物理量是树的叶子节点，分别对应一条时间序列．其中，每个设备相互独立，序列信息也单独管理．然而，根据我们对实际用户场景的调研，大量设备具有相似的职能，并且拥有一致的时间序列信息，因此我们针对性地优化了设备的组织管理方式．如图 2（b）所示，我们将若干设备分为一组，称为设备组．我们称设备组内设备数量的上限为设备组大小，也即图中的参数M．同一个设备组内含有若干不同设备，它们共享完全一致的物理量信息．这样，设备组内的设备不需要多次存储相同信息，而是共享物理量信息，减少了冗余元数据．

针对需要分组的多个设备，我们采用朴素的分组方法，分为两步：（1）分析各个设备下的物理量信息，将物理量信息完全一致的设备放入同一个集合；（2）根据设备组的大小，也即组内的设备数量的上限，将每个集合划分为若干个设备组．这样，在相同的设备组中，所有设备的相同物理量名称的时间序列具有完全一致的元数据信息，在存储时可以共用重复的元数据，以此来减少冗余元数据的存储．

该分组压缩方法需要解决一些问题．首先是数据的存储问题，也即如何存储多个设备的相同物理量名称的时间序列；其次是数据索引问题，也即如何确认所存储的序列片段属于分组前的哪个设备．我们将在 3.2 节介绍压缩方法，解决数据的存储问题；接着在 3.3 节介绍二级索引方法，解决数据的索引问题．

3.2　短序列的拼接压缩方法

在我们的分组模型中，若干相似的设备被分在同一设备组内，进行统一管理．得益于我们的分组方法，同一设备组内的同名物理量具有良好的性质，它们具有完全一致的特征，包括数据类型、编码方法、压缩方法等，但是时间戳可能并不对齐．基于这些良好的性质，我们提出时间序列拼接的压缩方法，进一步提高存储效率，解决短序列压缩效率低的问题．

如图 3 所示的例子，现在需要存储两个设备，分别是 device1 与 device2．我们发现两个设备都具有两个相同物理量名称的时间序列，分别为 s1、s2．在不分组的存储模型中，需要存储四条时间序列，对应 8 个数组，它们之间的元数据信息互不共享，存储单元相互隔离．在使用我们的分组压缩方法后，TsFile 可以识别出这两个设备的相似性，将这两个设备放入同一个存储组 group1．对于这个设备组内的序列，我们采用同名物理量拼接的方法来压缩．相同物理量名称的时间序列的拼接与存储的具体操作如下：（1）选定某个同名物理量，例如 s1；（2）依次遍历该设备组下所有设备，分别为 device1 和 device2，取出该物理量的时间戳列与数值列；（3）依照遍历次序，分别拼接时间戳列和数值列；（4）在存储到 TsFile 文件时，对 s1、s2 分别应用各自的编码或压缩方法．比如，数据类型为 32 位有符号整数的 s1 可以采用默认的二阶差分编码，并且使用 LZ4[20]方法对编码后的数据进行压缩，而数据类型为 64 位浮点数的 s2 可以采用默认的 Gorilla[16]方法进行编码，然后使用 LZ4 方法对编码后的数据进行压缩．按照这个流程，就可以拼接所有相同物理量名称的时间序列．这样，就把多个设备的相同物理量名称的时间序列拼接为了设备组内的一条时间序列．而且，该分组压缩方法并没有改变时间序列采用的编码与压缩方法，只是将多个短序列相互隔离的编码、压缩过程转化为了一个长序列的编码、压缩过程．

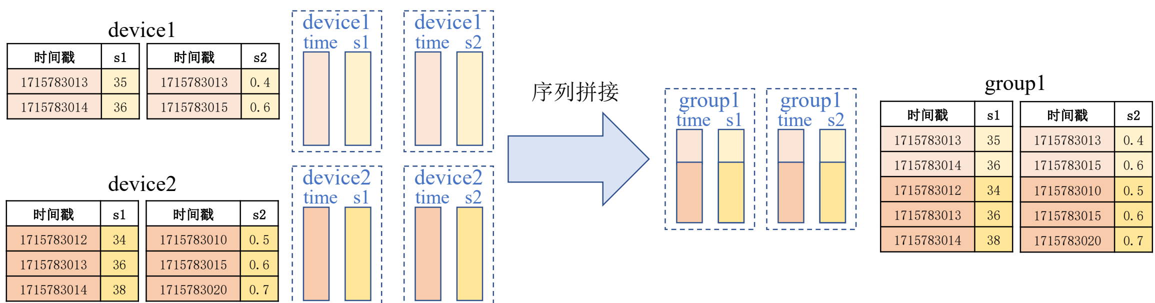


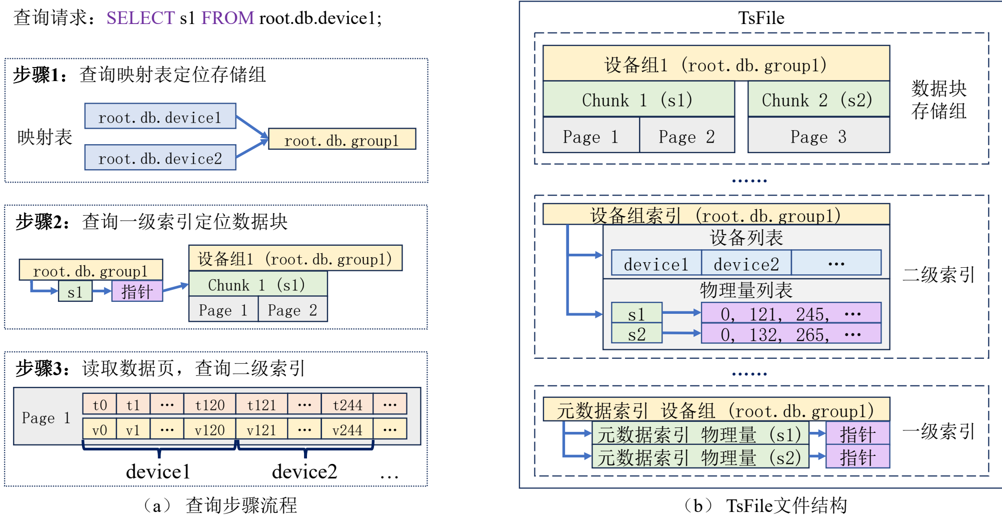
图 3 设备组中相同物理量名称的时间序列拼接存储

在序列数量众多、短序列频繁出现的海量序列场景下，该分组方法能有效降低实际存储序列单元的数量，并且通过将短序列拼接为长序列的方法有效提高编码与压缩效率. 相同物理量名称的时间序列拼接方法解决了海量序列场景下数据页压缩比高的难点. 由于我们将多个短序列拼接为长序列，压缩方法更容易识别出数值列或时间戳列的重复模式，能够降低压缩比，提高存储效率. 此外，该方法可以充分利用 TsFile 的存储结构. 对于 TsFile 存储引擎来说，存储一条时间序列的主要信息为：序列的元数据信息（名称、数据类型等）、时间戳列、数值列；经过我们的相同物理量名称的时间序列拼接方法，可以得到一条新的时间序列，它包含的主要信息也是这三者，可以直接通过 TsFile 存储引擎进行存储. 该方法自然也需要存储额外的信息，比如在图 3 的例子中，如果仅仅知道 2 条拼接后的时间序列，是无法还原出拼接前的 4 条时间序列的，因此我们还扩展了索引方法，来定位每行数据属于哪个设备，这部分内容将在 3.3 节详细介绍.

3.3 两层索引结构与查询重写

在使用分组压缩方法后，TsFile 中设备的相同信息被充分利用，但是对应的也丧失了独立性. 在原本不分组的 TsFile 查询框架下，不同设备的存储相互隔离，因此可以通过索引快速确定设备数据块的存储位置；而在我们提出的分组压缩方法下，多个设备共用了数据块，需要额外的信息来辅助定位. 因此，我们将原本的一层索引结构扩展为两层索引结构，并设计了查询重写方案，来保证分组压缩方法的有效性. 在本小节中，我们首先介绍查询重写方案的工作流程，接着阐述其中索引的具体实现结构，并分析该方法成功的原因.

查询重写方案主要依赖于两类数据结构来运行：（1）其一是文件所包含的设备到设备组的映射表，它能快速确定设备所在的设备组；（2）其二是两层索引结构，一级索引可以定位数据块的位置，二级索引可以定位数据页中数据点的位置. 我们在图 4（a）中给出一个完整的查询示例.



（a）查询步骤流程 （b）TsFile文件结构

图 4 查询重写示例与扩展的 TsFile 文件结构

在图 4（a）的例子中，用户想要查询设备 root.db.device1 的物理量 s1，并且发送了相应的 SQL 语句. 在接受到查询语句后，第一步是要完成设备的定位，也即需要确定设备 device1 位于哪一个设备组中. 利用映射表，可以快速定位设备 device1 所在的设备组为 root.db.group1. 这样，原本针对设备 root.db.device1 的查询就被重定向到设备组 root.db.group1 了. 第二步是找到设备组的存储位置，为了做到这一点，需要使用二级索引定位对应的时间序列. 在本示例中，所需要查询的是物理量 s1，通过一级索引中查找到的文件指针，可以快速找到该设备组中时间序列 root.db.group1.s1 的位置，获取其数据块. 当得到数据块后，就可以读取其中的数据页了. 第三步是在数据页中定位所查找的设备的数据点. 数据页中包含两个列表，其一是时间戳的列表 t，

其二是数值的列表 v. 通过二级索引，可以确认设备组 group1 内包含若干个设备，分别是 device1、device2 等；还可以确定该设备组内全部的物理量，包括 s1、s2 等. 可以找到物理量 s1 对应的数据点索引，在本示例中，所查找到的数据点索引为：$\{0,\ 121, 245,\ \cdots\}$，它表示了各个设备的第一个数据点的索引. 比如，第一个设备 device1 的数据点索引为 0，第二个设备 device2 的数据点索引为 121，因此可以确定设备 device1 的数据点是 (t0, v0) 到 (t120, v120). 以此类推，就可以确定各个设备的数据点位置. 反之，当读取到任意一个数据点，也可以根据该数据点索引找到其所属的设备. 最后，处理这 121 个数据点，就可以得到查询的结果.

为了实现该查询重写，我们需要改进 TsFile 的文件结构，使它支持上述提到的两个数据结构. 其中，映射表可以根据二级索引重建，因此并不需要存储到文件中；只需要存储二级索引即可. 我们将增加的二级索引数据保存在文件元数据（TsFile Metadata）区，如图 4（b）所示. 二级索引包含了文件中所有的设备组的结构信息，存储了组内设备的列表和组内物理量的列表，每个物理量会指向一个数据点索引. 这样，在我们的分组压缩方法下，索引结构变为了两层. 其中第一级是设备组中各时间序列的索引，依赖于 TsFile 原生支持的索引结构实现；第二级是设备组中各物理量的数据点索引列表. 利用这两层索引，我们能快速确定任意一条序列的某段数据在文件中的位置.

使用分组编码方法将索引的结构划分为两层，并没有减少索引的数量，但是降低了存储开销，其主要原因在于降低了字节数更多的字符串类型的数据存储量. 我们将通过公式的推导来确认这一结果，所用到的符号及其含义列在表 1 中. 其中，设备名称、物理量名称、索引的指针分别用d、s、p表示，时间序列的完整路径通过设备名称与物理量名称添加英文句点拼接得到，也即$d.s$；对于各个变量存储时所占用的平均长度，我们使用字节数来衡量，使用符号L；对于设备的数量，使用符号N_d，对于物理量的数量，使用符号N_s；对于设备组的大小，也即设备组内设备数量的上限，采用符号M.

表 1 公式中各符号的含义

名称	符号	含义
设备的名称	d	一个描述了设备完整路径的字符串
物理量的名称	s	一个描述了物理量名称的字符串，$d.s$构成完整路径
索引的指针	p	一个整数，表示了数据在文件中的位置
设备名的平均长度	L_d	设备名称字符串的平均字节数
物理量名称的平均长度	L_s	物理量名称字符串的平均字节数
索引指针的平均长度	L_p	索引指针的字节数
设备组大小	M	每个设备组内至多有多少个设备
设备总数	N_d	文件中设备的数量
同名物理量总数	N_s	每个设备下物理量的总数

时间序列的索引一般分为两部分，其一是完整的序列路径，也即将设备名称与物理量名称拼接起来，得到$d.s$，在 TsFile 中存储为字符串；其二是索引指针p，一般是一个 INT 类型的整数，指向了文件中某个数据块的位置. 这样的二元组$(d.s, p)$就可以定位某条序列的数据在文件中的位置，而索引的存储开销就是这样的二元组的总字节数. 这里为了方便计算字符串的总长度，统一使用了字节数的平均值. 我们假定文件中含有N_d个设备，其中每个设备都有且仅有N_s个物理量，使用设备名、物理量名称、索引指针的平均长度来衡量索引的存储开销. 我们将证明，在设备组大小M足够大时，真实场景下分组方法的索引开销更小.

引理 3.1. 不分组方法中索引存储开销与时间序列数量、设备名、物理量、索引指针的平均长度成正比.

文件中含有N_d个设备，其中每个设备都有且仅有N_s个物理量，因此共有$N_d \cdot N_s$条时间序列. 如果使用不分组方法，则需要对每个序列都存储完整的路径，并且保存一个文件指针，因此总的存储开销就是序列数乘以索引二元组的字节总长，也即$N_d \cdot N_s \cdot (L_d + L_s + L_P)$字节.

引理 3.2. $N_s > 1$, $M > 2$时分组方法中设备名称与物理量名称的存储开销小于不分组方法；分组方法的索引指针存储开销总是大于不分组方法.

使用分组压缩方法后，索引共有两层，并且产生了$\lceil N_d/M \rceil$个设备组. 其中，第一层索引需要记录每个设备组的序列，因此需要记录的序列数量减少了，仅需要记录$\lceil N_d/M \rceil \cdot N_s$条序列，总开销是$\lceil N_d/M \rceil \cdot N_s \cdot (L_d + L_s + L_P)$字节；第二层索引需要同时记录设备列表、物理量列表、数据点索引. 其中，设备列表的总开

销为$N_d \cdot L_d$字节，物理量列表的总开销为$\lceil N_d/M \rceil \cdot N_s \cdot L_s$字节，数据点索引的总开销至多为$\lceil N_d/M \rceil \cdot N_s \cdot M \cdot L_p$字节，第二层索引的总开销为$\lceil N_d/M \rceil \cdot N_s \cdot (M \cdot L_p + L_s) + N_d \cdot L_d$字节. 因此，分组压缩方法的总开销为两层索引开销的总和，至多为$\lceil N_d/M \rceil \cdot N_s \cdot (L_d + 2L_s + (M+1)L_P) + N_d \cdot L_d$字节. 我们将开销的情况对比整理到表 2 中. 只要满足不等式$N_s \cdot M > N_s + M$，分组压缩方法中设备名称的存储开销就更小；只要满足$M > 2$，分组压缩方法中物理量名称的存储开销就更小；分组压缩方法总是有更多索引指针，开销增加约$N_{\mathrm{d}} \cdot N_s \cdot L_P/\mathrm{M}$. 根据变量的具体含义，$N_s > 1$一般成立，此时命题$N_s \cdot M > N_s + M$含于命题$M > 2$.

表 2 两种方法索引的存储负担

名称	符号	不分组方法索引开销	分组压缩方法索引开销	近似差值
设备名的平均长度	L_d	$N_d \cdot N_s \cdot L_d$	$(\lceil N_d/M \rceil \cdot N_s + N_d) \cdot L_d$	$\left(\left(1-\frac{1}{M}\right)N_s - 1\right)N_d \cdot L_d$
物理量名称的平均长度	L_s	$N_d \cdot N_s \cdot L_s$	$2\lceil N_d/M \rceil \cdot N_s \cdot L_s$	$\left(1-\frac{2}{M}\right)N_d \cdot N_s \cdot L_s$
索引指针的平均长度	L_p	$N_d \cdot N_s \cdot L_p$	$\lceil N_d/M \rceil \cdot N_s \cdot (M+1)L_P$	$-\left(\frac{1}{M}+\frac{M+1}{N_d}\right)N_d \cdot N_s \cdot L_p$

推论 3.3. **与不分组方法相比，分组方法是否减少了索引大小和设备数量N_d无关.**

我们可以发现表 2 中，不分组方法的索引开销与分组压缩方法的索引开销各项都含有正整数N_d项. 因此，三项同时除以N_d项不影响它们之间的大小关系，故推论成立.

定理 3.4. **当$\left(\left(1-\frac{1}{M}\right)N_s - 1\right)L_d + \left(1-\frac{2}{M}\right)N_s \cdot L_s > \left(\frac{1}{M}+\frac{M+1}{N_d}\right)N_s \cdot L_p$时，分组方法索引存储开销少于不分组方法.**

我们可以通过表 2 中的近似差值得到该结论. 这里，近似差值指的是不分组方法索引开销减去分组压缩方法索引开销的估计. 由于不等式$N_d/M \le \lceil N_d/M \rceil < N_d/M + 1$成立，我们可以对各项索引的大小进行放缩：我们缩小了设备名和物理量名称减少的开销估计，并且放大了索引指针的开销估计，然后作差得到定理中的不等式. 在实际情况下，定理中的不等式一般成立. 这主要有两个原因：(1) 不等式左侧与设备组大小M相关的常数项系数更大；(2) 字符串一般比整数占用更多的存储空间，也即不等式左侧与字节数相关的参数值更大. 为了方便理解，我们在样例 3.1 中给出一个实际样例.

样例 3.1. 估计分别使用分组压缩方法和不分组方法时，某车辆数据集的索引存储开销. 假设我们需要存储的设备数为$N_{\mathrm{d}} = 6700$，每个设备有$N_s = 27$条时间序列. 每个设备名称平均需要$L_d = 72$字节来存储，物理量名称平均需要$L_s = 8$字节来存储，而整数索引需要$L_p = 4$字节来存储. (1) 使用原本的方案，则需要分别存储$6700 \times 27 = 180900$条序列的索引，消耗$180900 \times 80\mathrm{B} = 14472\mathrm{KB}$空间；索引指针的存储需要$180900 \times 4B = 724\mathrm{KB}$空间. 最终不分组方法合计消耗 15195KB 空间. (2) 在使用分组压缩方法后，假设每个组至多存储$M = 100$个设备信息，则合并后有$6700 \div 100 = 67$个设备组. 其中，第一级索引需要记录$67 \times 27 = 1809$条序列，字符串存储开销为$1809 \times 80\mathrm{B} = 145\mathrm{KB}$，索引指针的存储开销为$1809 \times 4\mathrm{B} = 7\mathrm{KB}$. 在第二级索引中，需要存储全部 6700 个设备的设备列表，开销为$6700 \times 72 = 482\mathrm{KB}$；还需要存储$67 \times 27 = 1089$条序列的物理量列表，开销为$1809 \times 8B = 14\mathrm{KB}$，数据点索引指针的开销为$1089 \times 100 \times 4\mathrm{B} = 756\mathrm{KB}$. 最终分组压缩方法的合计消耗 1372KB 空间，减少了 91%. 该理论估计与实验结果非常吻合.

分组压缩方法能有效降低索引的大小，这得益于分组策略充分利用了大量设备的相同特征，降低了时间序列名称的存储量. 此外，在上述推导中，我们默认了每条序列的一级索引只有一个文件指针. 在实际的 TsFile 中，每个时间序列的文件指针数量和它的数据页数量有关，这样能在数据量多的情况下使用多个索引快速定位数据. 然而，根据对真实世界数据集的分析，我们发现短序列场景中的数据量很小，一般不会超过 200 行，这就使得它们在文件中只有一个数据页. 同时，在设备组大小M不过大时，拼接后的长时间序列也仅有一个或两个数据页，因此实际情况和理论推导非常吻合. 即便面临短序列更长一些的场景，一级索引中包含多个文件指针，所增加的指针存储开销也远小于分组方法减少的字符串存储开销，因此分组方法依然能有效减少索引大小. 不过，如果存储的序列非常长，一个 TsFile 内设备、时间序列的数量都很少时，定理 3.4 中的不等式可能并不成立. 对于这种情况，不分组的方法索引开销会少于分组方法的索引开销，而此类不含

短序列的场景不适合使用分组压缩方法，我们会在本文的第 4 节讨论这种情况.

4 短序列文件的合并

我们已经详细介绍了分组压缩方法，阐述了模型设计的理念、短序列的拼接压缩方法、如何构建索引并重写查询. 分组压缩方法能够很好地针对短序列大量存在的场景进行优化，但是该方法需要引入额外的处理开销，例如对序列分组的时间开销、查询重写的额外时间开销等等. 当数据库管理系统能将短序列合并为长序列时，短序列的数量会逐渐减少，存储劣势也不再明显，此时便逐渐不再需要分组压缩方法. 因此，我们将在本节详细介绍短序列文件的产生、合并过程，以及如何在合适的时机选择分组压缩方法.

4.1 短序列文件的产生与数据同步

我们通过分析企业合作伙伴提供的真实世界数据集，发现在很多场景下会存在大量短序列. 与长序列相比，短序列只有较少的数据点，这导致其存储效率很低. 短序列产生的原因有多种，常见的原因有：（1）数据同步周期短，导致文件序列只包含较短时间窗口内的数据；（2）海量序列场景下，并发请求多，数据刷盘（flash）频繁，导致大量短序列存储于同一文件. 我们在这里给出两种场景的详细介绍.

场景样例 1. 端边云协同. 在实际使用场景中，往往需要端侧、边侧和云侧协作来完成数据的收集、清洗、管理与分析. 举例来说，使用 TsFile 文件格式的 Apache IoTDB 通过分布式结构进行数据管理，并且能在云侧和边侧灵活部署，利用云端同步工具无缝连接. 某公司旗下有多家工厂，每个工厂有若干车间，此时可以将 IoTDB 贯穿公司、工厂、车间三级物联网平台，实现设备统一联调联控. 在这种场景下，车间端侧的物联网设备负责采集数据，可以部署 IoTDB C++版本的客户端，在硬件资源受限的条件下采集数据；工厂层收集端侧数据并完成清洗工作，并且可对数据做业务分析；而公司云侧则可对工厂设备进行监控与分析. 端侧物联网设备 24 小时不间断地采集数据，但是为了数据能及时同步，同步周期可能很短，例如每分钟同步一次. 此时，该设备每分钟的数据在同步文件内只包含这一分钟内的数据点，数量较少，因此 24 小时内产生的大量数据被切分到 1440 个碎片文件中. 边侧数据节点在接受到这些碎片文件后，需要重新整理、合并.

场景样例 2. 海量序列. 在海量序列场景中，时间序列数量可能非常庞大. 例如，某物联网公司使用 Apache IoTDB 记录了超过 700 万台某品牌汽车在实际运行过程中的 27 项指标,时间序列总数超过 2 亿条. 尽管使用了若干分布式节点，每个节点仍然需要处理大量时间序列的写入请求. 一般情况下，IoTDB 在内存中会缓存每条序列写入的数据，在数据量足够多时再刷盘到磁盘. 然而，在海量序列场景下，尤其是在请求的高峰期，每条序列分配到的平均缓存数量小，因此在缓存占满后不得不将数据立即刷盘到磁盘中. 因此，写入到同一个 TsFile 中的序列数量众多，但每条序列都很短，平均仅有 80 个数据点，对应的时间跨度约为 4 分钟.

在上述两个场景样例中，都存在大量的短序列，但是产生的原因与带来的问题略有区别. 在场景样例 1 中，存在数据的同步问题. TsFile 的数据同步由数据库管理系统的具体实现决定，例如在 Apache IoTDB 中，数据同步的方式取决于用户的配置，主要有两种：（1）批量同步（batch），该同步模式下会通过 TsFile 文件的传递完成；（2）流式同步（stream），一般情况下发送同步任务，例如发送写入计划来保证新增数据的同步，并在任务积压时采用发送 TsFile 的方法同步更多数据. 简单来说，TsFile 的文件格式的数据同步基本单位是数据本身而不是单个 TsFile ，文件格式并没有限制数据同步的方法，数据同步具体使用的方法取决于用户的配置. 在场景样例 1 中，端侧产生的时间序列数据不间断地产生时序数据，但是在同步时数据点被切分在多个短序列中在不同时间进行传输，边侧会接收大量碎片文件，需要进行整理与合并.

在场景样例 2 中，短序列直接在服务器本地产生，不存在端边云协同场景的定时同步问题，但是存在副本之间的一致性维护问题. 在场景样例 2 中，用户为每个 IoTDB 数据节点（Data Node）都配置了 2 个副本，两个数据节点互为备份. 此时，两个节点主要通过同步写入计划的方法同步新增的数据，同步的过程并不直接使用 TsFile 文件格式. 该场景中需要解决的主要问题是本地的文件中存在大量短序列，存储压力大. 在该场景中，当缓存不足、必须将内存中的数据刷写到磁盘时，数据库管理系统无法在短时间内读取历史数据并

整合为长序列，直接将缓存中仅含有少量数据的时间序列刷写到磁盘，导致 TsFile 中含有大量短序列.

4.2 短序列碎片文件的合并

TsFile 文件的合并目标主要包括三点：（1）整合乱序的数据，避免查询时进行临时的顺乱序归并排序，提升查询性能；（2）整合相同时间序列的数据，增加数据块的大小，避免多次查询和读取数据块，提升查询性能；（3）减少文件的数量，这有一定的益处. 例如，在使用 TsFile 文件格式的 IoTDB 中，每个文件对应了文件资源（TsFileResource）内存结构，更少的文件数量就对应更少的内存占用. 随着合并过程的进行，存储了大量短序列的碎片文件会被合并成为含有长序列的 TsFile 文件.

和数据的同步类似，TsFile 的合并也依赖于数据库系统的具体实现. 例如，在 IoTDB 中 TsFile 的合并过程可以简单分为三个阶段：（1）文件的选择，选出需要被合并的文件；（2）合并调度，根据当前的系统状态选择合并任务；（3）执行合并，读取待合并的文件，在内存中排序、整理、合并，然后写入到磁盘中的新文件. TsFile 针对乱序数据的存储采用顺乱序分离的策略：一般情况下，TsFile 中一条时间序列下存储的数据点都是时间升序的，不存在乱序数据；乱序数据被额外记录在乱序文件中. 在短序列碎片文件的合并中，乱序数据并不是我们的研究重点，因此我们本节主要考虑顺序文件的合并. 这里所说的合并短序列文件，主要指的是顺序的 TsFile 文件，也即同一序列下数据点都是按照时间升序排列的.

不分组的 TsFile 文件的合并执行过程如下：（1）读取待合并文件到内存，并且锁定对应的文件；（2）依据系统配置中的规则，记录合并信息，例如合并的类型、待合并文件、合并的目标文件等；（3）将待合并文件中的相同时间序列的不同片段做排序、合并，得到新的时间序列；（4）将合并后的时间序列写入目标文件中；（5）在内存中用新文件的文件资源替换待合并文件的资源，释放对待合并文件的锁，将待合并的文件删除. 若有必要，最后删除合并日志.

使用分组压缩方法的 TsFile 文件合并执行过程仅在第 3 步不同，由于分组方法将多个设备的时间序列压缩到同一个设备组中，因此第 3 步需要改写如下：（3.1）将待合并文件中设备组的时间序列切分为多个设备的时间序列，相当于将拼接后的长序列解压缩为若干个短序列；（3.2）将解压缩后的时间序列的不同片段做排序、合并，得到新的时间序列；（3.3）对合并后的时间序列，重新应用分组压缩方法.

此外，如有必要，上述（3.3）步也可以不使用分组压缩方法，这样可以读取若干待合并的分组压缩 TsFile 碎片文件，最终合并得到不分组的 TsFile 文件. 这种处理方法在短序列逐渐消失、不再需要使用分组方法时很有效，我们将在 4.3 节中详细论述分组方法在短序列合并过程中的应用.

4.3 分组压缩方法在短序列合并过程中的应用

在我们的方法中，只要有足够设备拥有名称、数据类型相同的物理量，就可以通过分组压缩方法充分剔除冗余的元数据，提高存储效率. 该分组压缩方法减少的存储开销主要包括：（1）将短序列拼接为长序列，提高数据页的压缩效率，减少存储开销；（2）建立二级索引，删除一级索引中冗余的序列元数据信息. 而这两项优化都需要 TsFile 内存储的数据具有一定的特征：前者需要数据中存在大量相同数据类型的短序列；后者要求文件内序列的数量足够多，设备组大小足够大，也即有足够多的设备共用相同的元数据信息. 因此，当数据不再满足这两类特征时，应该根据实际情况进行选择，采用最高效的存储方法. 对于两个场景样例，分组压缩方法的实际应用分别展示在图 5、图 6 中.

在场景样例 1 中，我们介绍了端边云协同的场景. 在该场景样例中，分组压缩方法可以在数据同步与云侧存储发挥重要作用，如图 5 所示. 端侧的数据同步产生了大量碎片文件，每个文件中序列的数据点数都很少，此时不分组存储的 TsFile 数据占用了大量空间；这些等待同步的碎片文件发送到边侧后，可以采用分组压缩的方法进行高效存储，减少文件的大小；边侧的数据量随着时间不断增加，短序列逐渐增多，此时便需要进行文件合并. 边侧可以将碎片文件合并为分组压缩的 TsFile，随着时间的推移，每个端侧设备都在不间断地产生数据，这些序列都将在边侧逐渐合并为长序列. 由于每个 TsFile 的大小有一定上限，因此在序列变长后，一个文件可能不再能容纳多个设备的数据，此时需要将这些设备分别存储在不同的 TsFile 中. 这样，

随着碎片文件的合并，单个 TsFile 内存储的序列会逐渐减少，序列长度也会逐渐增加，到达一定程度时便不再适合使用分组压缩方法. 此时，就可以使用不分组方法的 TsFile 存储长序列内容；最终，云侧使用不分组方法存储了各个设备的长序列. 此外，该过程中碎片文件的合并在云侧也可能发生，例如边侧新增的数据尚未合并到长序列，但是接收到同步请求，碎片文件可能被同步到云侧，云侧就可能在合适的时机将碎片文件整合到不分组存储的长序列文件中.

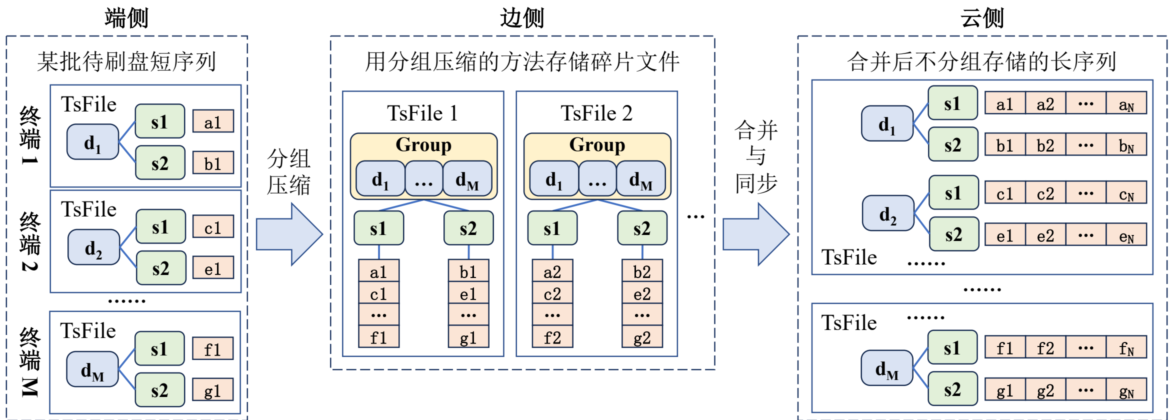


图 5　端边云协同场景中分组压缩方法的应用

在场景样例 2 中，我们介绍了海量序列场景. 在该场景样例中，短序列产生的主要原因是内存不足导致的频繁数据刷盘. 针对这一问题，分组压缩方法可以在数据刷盘、存储短序列中发挥重要作用，如图 6 所示. 当缓存不足，必须将内存中的数据刷写到磁盘时，可以使用分组压缩方法完成写入，该方法的优势在于减少了磁盘写入次数、同时降低了写入数据量. 由于分组压缩方法一次处理多个设备，因此能减少文件的写入操作；同时该方法降低了压缩比，减少了数据页与索引的大小，减少了写入的序列化数据. 此外，该场景中内存的压力较大，而分组压缩方法引入的额外开销少，能有效保证性能. 这是因为分组压缩方法充分利用了时间序列相似的元数据信息，于是在内存中设备组内的元数据得到了重复利用，同时对短序列做拼接存储能极大提高存储效率. 在写入请求的低谷期，服务器有足够的空闲算力对文件进行合并，在合并的过程中会逐渐将短序列合并为长序列. 与场景样例 1 的分析类似，TsFile 的合并过程会逐渐减少同一文件中序列的数量、增加每条序列的长度，到达一定程度时便不再适合使用分组压缩方法. 此时，就可以在合并时使用不分组方法的 TsFile 存储长序列内容.

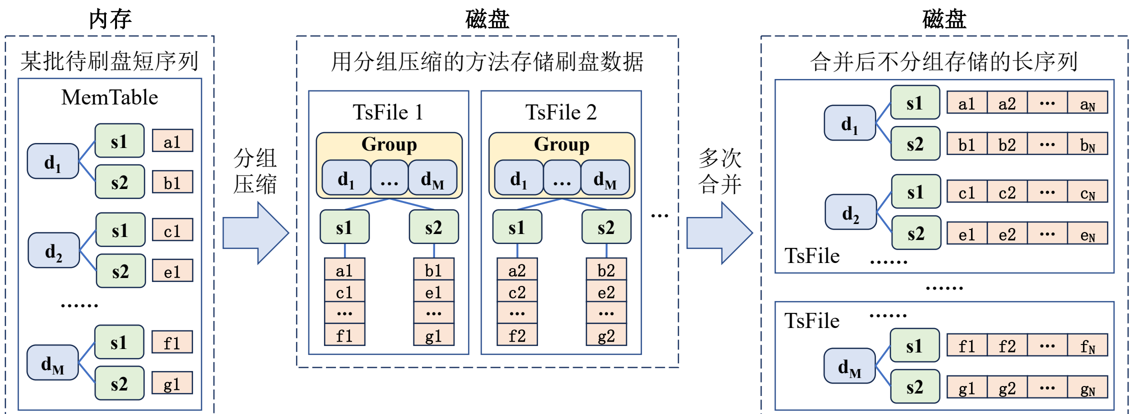


图 6　海量序列场景中分组压缩方法的应用

总的来说，依据短序列产生的具体情况，可以灵活地使用分组压缩方法，分组压缩方法可以在短序列存储、数据同步等场景发挥重要作用. 我们在实验部分也详细对比了分组压缩方法和不分组方法在各项指标上的表现，并且分析了序列合并过程对压缩效果的影响，分析了应该在何种情况下从分组压缩方法转变为不分组方法，综合考虑压缩效率与查询、写入等多方面的权衡.

5 实验分析

在本节中，我们将通过以下几个方面对 TsFile 分组压缩方法的性能进行分析：（1）空间压缩比；（2）读取性能；（3）写入性能；（4）碎片文件的合并性能. 我们选用 Apache IoTDB 作为数据库管理系统，针对 Apache TsFile 的分组压缩方法展开实验. 我们使用的实验配置是：24 核 Intel(R) Core(TM) i9-13900HX CPU@2.20 GHz，16GB 内存，1TB 固态硬盘.

5.1 实验设计

我们在 2 个真实世界数据集上验证了所提方案的有效性. 表 3 给出了数据集的基本信息. 其中，车辆数据集是大量某品牌的汽车在实际运行过程中采集的数据，总设备数超过 700 万台，时间序列数量超过 2 亿条，主要数据类型为 32 位有符号整数和 64 位浮点数，本实验截取了其中部分设备在某段时间内的真实数据；船舶数据集是在船舶航行测试中采集的运行数据，主要数据类型为文本，存储了 json 格式的多项航行参数，本实验截取了其中部分设备在某段时间内的真实数据.

表 3 实验数据集

名称	平均每文件设备数量	平均每文件时间序列数量	总行数	主要数据类型
车辆数据集	6772	182844	14901948	INT32, DOUBLE
船舶数据集	68018	136036	408108	TEXT

我们使用各场景下采集到的真实数据集进行实验：我们将这些数据通过分组压缩方法进行整理，重新写入到 TsFile 中，与不分组的 TsFile 进行比对. 我们的实验主要涉及三大方面：（1）文件的压缩效率；（2）文件的读取与写入效率；（3）碎片文件合并中的耗时以及压缩效率的变化.

通过比对两个文件的大小，可以直观地得到该方法的压缩比，这里我们压缩比的定义为：压缩后的文件大小除以压缩前的文件大小. 因此，这个比值一般是一个小于 1 的数字. 同时，我们还详细分析了文件结构中各部分的占比，可以进一步解释分组压缩方法是如何提升压缩效率的；接着，通过对比不分组与分组压缩两种方法下文件的查询、写入速率等指标，可以分析出两类方法的文件读取与写入效率；最后，我们还设计了碎片文件的合并实验，在这个实验中数据集被进一步打散，分别存储在若干 TsFile 碎片文件中，以模拟短时间序列产生的场景. 通过对比不分组与分组压缩两种方法的文件合并耗时，可以分析出两种方法在碎片文件合并时的效率. 此外，我们还分析了合并过程中文件压缩效率的变化，正如在第 4 节中所说的，这有助于我们分析选择合适方法的时机.

5.2 分组压缩方法的压缩效率分析

针对各数据集，我们选用不同的设备组大小进行压缩，对比文件压缩效果. 一方面，这有利于我们选定合适的实验参数进行后续实验，另一方面，这有利于分析设备组大小对分组编码效率的影响. 我们不仅分析了设备组大小变化对文件整体的影响，也深入分析了文件的各项组成部分的变化，主要是数据页的压缩效率以及文件索引的大小变化. 我们首先评估了使用分组压缩方法后，文件总大小的变化，结果展示在图 7 中.

我们发现，即使设备组大小上限仅为 2，也即每个组内至多压缩两个设备，也能有效降低文件总大小. 我们采用压缩比来评估文件的压缩效率，其定义是使用分组压缩方法后文件的大小除以原文件的大小. 如图 7 所示，随着设备组大小的增加，文件的总大小逐渐降低. 当设备组大小到达 100 左右时，文件压缩比趋于稳定. 此外，在不同数据集上文件的最佳压缩比收敛于不同值，我们认为这和数据集的固有特征有关. 为了解释这一点，并且确认分组压缩方法的压缩效率是如何提升的，我们进一步分析了文件中数据页和索引的变化

情况.

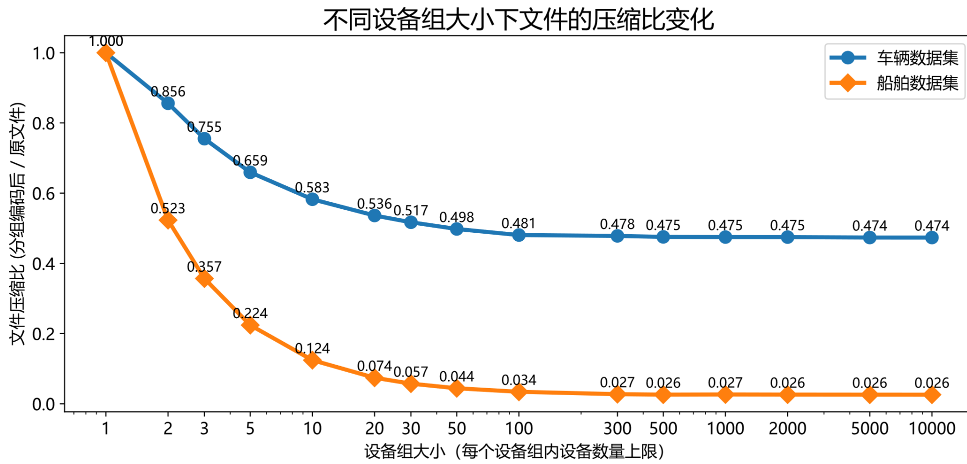


图 7　不同设备组大小下文件的压缩比变化

我们从数据页的压缩和索引的压缩两方面做了深入分析. 我们分析了在不同设备组大小时，每个文件中数据页的总大小以及索引的总大小. 由于不同数据集本身的大小不同，我们将总大小数据进行了归一化处理，也即分析压缩比变化：将分组后的数据页大小除以分组前的大小，得到一个小于 1 的压缩比，分析该压缩比的变化；对于索引的大小我们也做同样的归一化处理. 我们将数据页总大小的变化展示在图 8 中，将索引大小的变化展示在图 9 中.

从图 8 我们可以得知，使用分组压缩方法后，文件中数据页的大小下降了，其下降程度与具体数据集以及设备组的大小相关. 随着设备组大小的逐渐增大，数据页的压缩比逐渐下降，并趋于稳定. 在不同数据集上，压缩比下降并收敛到不同的数值，我们认为，这和数据集固有的数据类型、数据特征以及编码方式有关. 首先，船舶数据集中以字符串形式存储了航行日志，日志的格式非常统一，并且含有大量重复值；其次，实验过程中这些字符串通过字典编码的方式存储在 TsFile 中. 因此，字典编码充分利用了这些日志的重复特征，在数据分散于大量短序列时，每条短序列都需要独立存储一个字典信息，存在大量冗余；当分组压缩方法拼接短序列后，多个设备的数据共用同一个字典信息，显著提高了编码效率，因此船舶数据集的最终压缩比更小. 而车辆数据集主要包含整数与浮点数，短序列拼接后带来压缩比的降低有限.

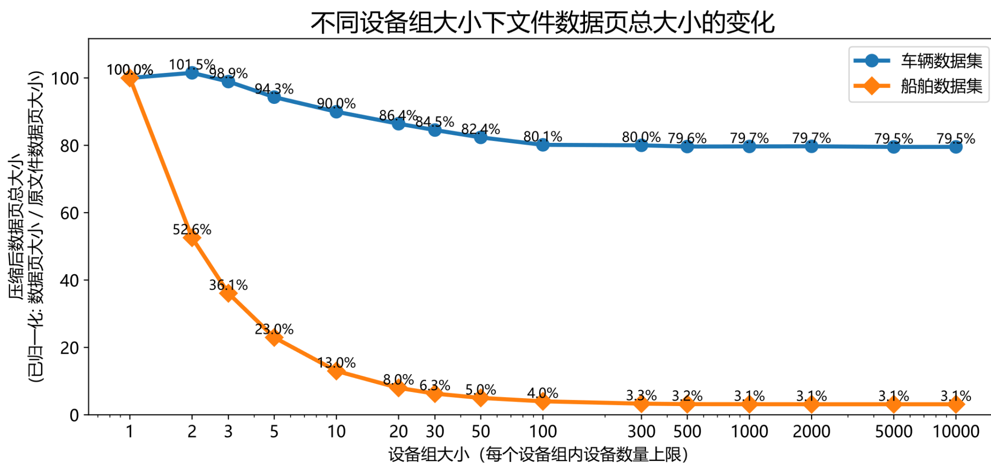


图 8　不同设备组大小下数据页总大小的变化

从图 9 可以看到，在不同数据集上，随着设备组大小的增加，文件的索引大小也在逐渐降低，最后趋于稳定，这说明我们的分组压缩方法能有效减少索引的存储开销. 此外，我们可以看到设备组大小为 100 时，车辆数据集索引的压缩比约为 8.7%，索引大小降低了约 91.3%，这和我们在样例 3.1 中计算的理论值非常吻合.

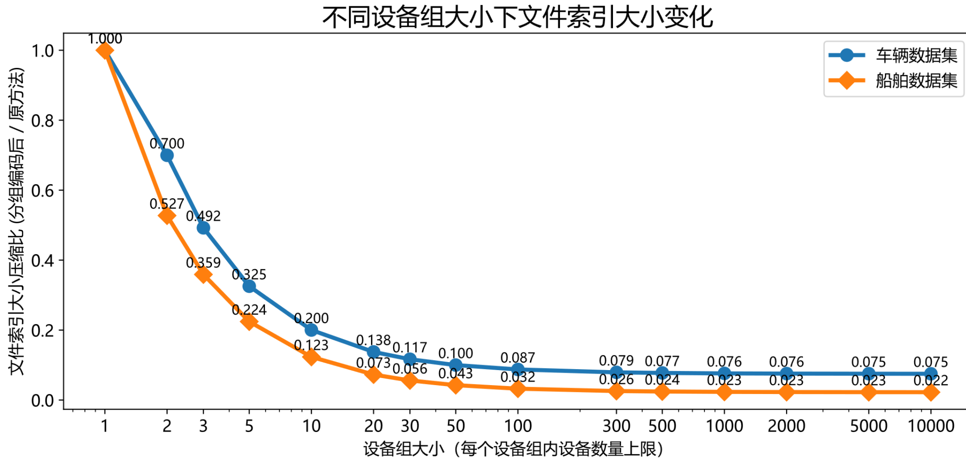


图 9　不同设备组大小下索引总大小的变化

此外，分组压缩方法可以降低索引的存储占比. 我们分析了索引占文件总大小的变化情况，结果展示在图 10 中. 如前文提到的，分组压缩方法将多个设备的元数据充分利用，但是并没有减少索引指针的数量或者数据页压缩前的大小，而是通过利用其相似性减轻了存储负担：对于数据页而言，我们通过拼接短序列为长序列的方式提高压缩比，降低存储成本；对于索引而言，我们将索引分为两层来减少冗余数据，降低存储成本. 然而，数据页和索引的大小都在变化，并且在不同数据集上呈现出差异. 因此，我们分析了索引大小占文件大小的比例，这样就可以得知在文件的存储中，是哪一部分占据了主导.

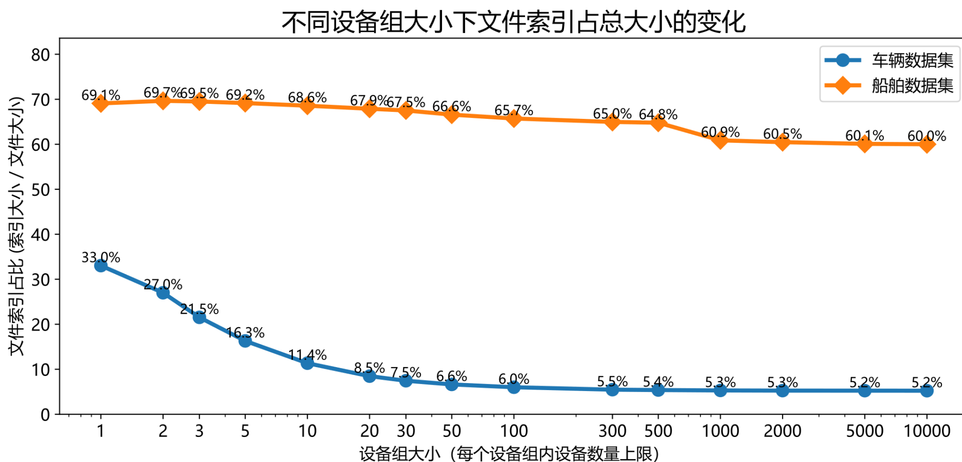


图 10 不同设备组大小下索引占文件总大小的变化

从图 10 可以看出，索引的存储占比随着设备组大小逐渐下降，这说明单位存储空间中留给数据页的部分在逐渐增多. 而不同数据集上索引的占比呈现明显差异，例如船舶数据集的索引存储占比明显较高，始终高于 60%；而车辆数据集的索引占比则从 33.0%下降到了 5.2%. 我们认为这是因为船舶数据集的数据特征导致

的. 从表 1 可以看到，尽管两个数据集在同一文件内都有十数万条序列，但是车辆数据集的行数有 1490 万条左右，船舶数据集的行数只有 40 万条左右，这说明车辆数据集中平均每条时间序列有约 109 个数据点，而船舶数据集平均每条序列只有约 3 个数据点. 尽管船舶数据集内每条序列的行数少，但是每个数据点的字符文本长度较长，平均长度约为 430 字节，而车辆数据集的每个数据点则只需要 4 字节（INT32）或者 8 字节（DOUBLE）来存储. 这样的差异导致船舶数据集存储在 TsFile 中时，元数据的占比会更高. 此外，在船舶数据集中，某些设备的 UID 信息会直接存储在序列名称中，这导致序列名称较长，也会反映到索引大小中.

总的来说，通过分析文件总大小的变化，我们确认了分组压缩方法能有效降低存储空间，在设备组大小约为 100 时能达到足够好的效果. 通过分析文件内各个部分的变化，我们验证了分组压缩方法的空间效率提升主要来自于两个方面：（1）序列索引的减少. 由于使用二级索引，该方法降低了所需存储的字符串数量，文件中对时间序列的索引存储开销降低；（2）序列压缩比优化. 由于将多个压缩比大的短序列拼接为压缩比小的长序列，数据块中数据页的压缩比进一步降低，节省了大量空间.

经过本节的实验，我们基本确认了在选中的数据集上，当设备组大小选为 100 时，分组压缩方法能够获得较好的压缩效果，并且能体现出该方法的特征. 因此在后续的实验中，我们统一选定设备组大小为 100 进行文件的查询、写入、碎片合并的分析.

5.3 查询效率分析

分组压缩方法的实现与 TsFile 文件格式相关联，不依赖于具体的数据库管理系统. 因此，我们主要分析在使用 TsFile 原生查询接口时查询的效率. 本实验测量了在固定查询次数下，不分组的 TsFile 与使用分组压缩方法的 TsFile 的查询耗时. 其中，查询的序列是从数据集中随机均匀地选取的，每次查询需要读取选中序列的全部数据. 为了进一步确认真实场景的查询效率，我们分别做无缓存和有缓存两种场景的查询实验. 其中，无缓存场景下每次查询都必须读取文件以获取查询结果；而有缓存的场景下，我们模拟了数据库管理系统的缓存实现，当缓存中含有需要查询的信息，则直接返回缓存中的历史查询结果，无需再次读取文件. 对于该缓存的大小，我们参考了 IoTDB 中的默认配置，在该配置下各个数据集都能全量缓存，也即缓存大小足以存储全部查询历史数据. 我们的实验结果如图 11、图 12 所示.

图 11 展示了车辆数据集中两种方法的 TsFile 查询耗时的比较，图 12 展示了船舶数据集中两种方法的 TsFile 查询耗时的比较，并且对比了不同序列长度对查询效果的影响. 图中横轴表示平均每条序列的数据点数，也即序列的长度. 比如在该场景下数据点数仅为 100 时，可以认为该序列为短时间序列. 当时间序列的长度逐渐增加时，查询消耗的时间逐渐增加，这是因为每条查询读取的数据点数量增加了.

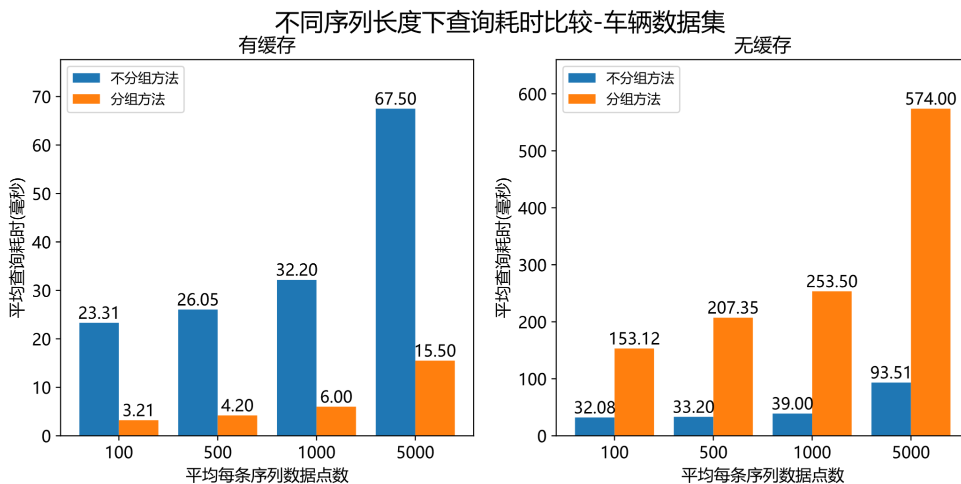


图 11　车辆数据集的查询耗时比较

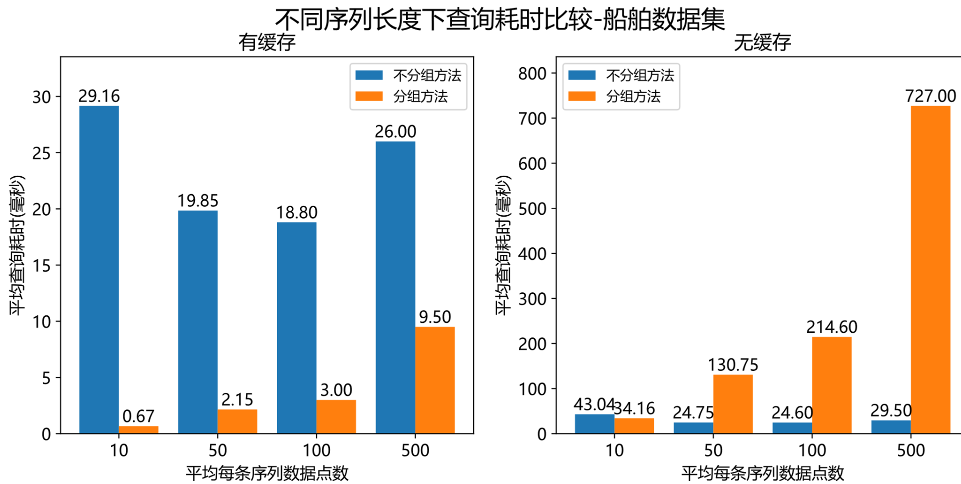


图 12 船舶数据集的查询耗时比较

在使用缓存时，分组压缩方法明显优于不分组方法. 这主要是因为，TsFile 的查询最小单元是数据页. 当接收查询请求后，TsFile 会根据索引定位数据块的位置，然后依次处理数据页，获取查询结果. 而分组压缩方法的优势在于，同一个数据页内存储了多个设备的时间序列数据，对应的，也会缓存多个设备的时间序列数据. 在本实验中，设备组大小为 100，这意味着读取使用分组压缩方法的 TsFile 后，每次都可以缓存至多 100 个设备相同物理量名称的时间序列数据，这极大地提高了缓存的命中率；而不分组方法每次读取 TsFile 后，仅能缓存该条时间序列的数据，因此缓存命中率相对低.

在不使用缓存时，每次处理查询请求都需要读取 TsFile 文件，因此各个方法的耗时都增加了. 相比于不分组方法，分组压缩方法的查询耗时会更多，这也与 TsFile 的查询最小单元是数据页有关. 由于分组压缩方法将多个设备的同名短序列拼接为长序列存储在数据页中，数据页的大小必定会变长. 而 TsFile 每次至少读取一个数据页，并且必须完整读取整个数据页，因此分组压缩方法总是需要读取更多字节的数据，于是耗时增加了. 而在不同数据集上，分组压缩方法耗时增加的程度不同. 我们认为这主要有两方面原因：(1) 如图 7 所示，船舶数据集的数据页压缩比明显低于车辆数据集，因此数据页的大小明显降低了，所以需要读取的字节数相对更少；(2) 数据页会限制每页中存储的数据量，在设置数据点数上限时，超过上限的数据会存储在下一页. 而车辆数据集的序列平均行数明显多于船舶数据集，这意味着将短序列拼接为长序列时，同一数据页内能容纳的序列数量更少.

总的来说，在有缓存的应用场景下，分组压缩方法的查询效率更高，这得益于数据页信息量大带来的缓存命中率的提高. 而在没有缓存的应用场景下，分组压缩方法的查询会变慢.

5.4 写入效率分析

如上一小节所述，分组压缩方法不依赖于具体的数据库系统，因此在本实验中，我们只使用 TsFile 原生写入接口测量写入效率. 本实验测量了相同数据，在分别使用不分组方法和分组压缩方法时，写入 TsFile 的耗时. 本实验的结果如图 13 所示.

可以从图 13 看到，分组压缩方法的写入耗时略低于不分组方法. 这主要有两方面的原因：(1) 调用写入接口的次数更少. 使用了分组压缩方法后，只需要调取更少次数的写入接口. 在得到需要写入的数据后，分组压缩方法会将同一设备组内的多个设备的同名时间序列做拼接，然后一次写入. (2) 数据压缩比更低，所需要写入的数据更少. 使用了分组压缩方法后，在内存中短序列会拼接为长序列. 根据前文对文件压缩效率的分析，长序列的压缩比更低，压缩后占用的空间更少，因此所需要写入的数据也就更少.

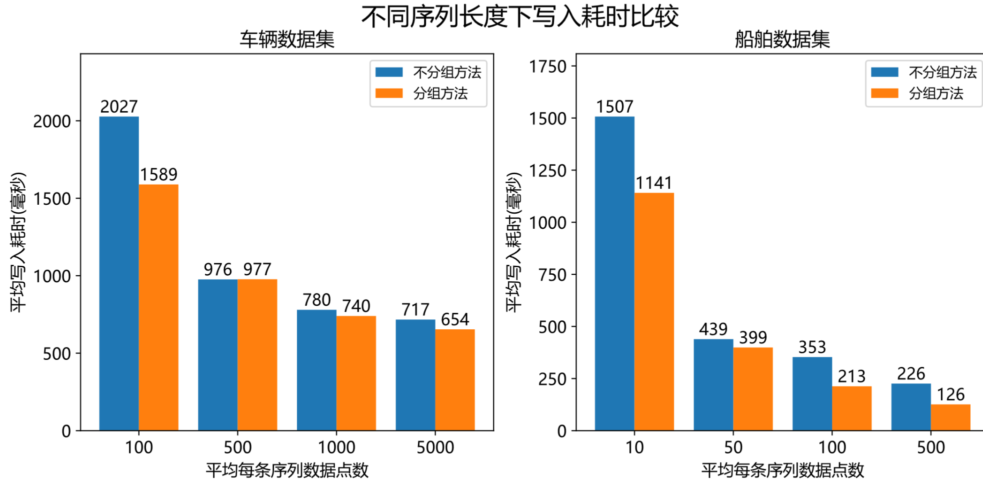


图 13 各数据集的写入耗时比较

总的来说，使用分组压缩方法写入短时间序列可以提高 TsFile 的写入效率.

5.5 TsFile碎片文件的合并

在本实验中，我们测量了分别使用不分组方法和分组压缩方法时，文件合并的效率. 我们将各个数据集上的不分组的 TsFile 文件打散为若干个碎片文件，以模拟端设备产生的大量小 TsFile. 接着，我们将碎片文件分别用不分组方法和分组压缩方法存储，为合并实验做准备. 最后，我们分别使用两种方法，合并对应格式的 TsFile 文件，测量其各个阶段的耗时. 由于数据集的时间序列层级结构不同，我们在表 4 总结了各个数据集被打散的情况.

表 4 不同实验数据集的打散情况

名称	限制文件内设备数	限制每序列行数	文件内序列数	打散后文件数量
车辆数据集	68	80	680	100
船舶数据集	681	3	1362	100

船舶数据集中每个设备下时间序列的数量较少，但是每个数据点占用的字节数更多；在碎片文件数量相同时，每个碎片文件中设备数量与时间序列总数更多. 我们测量了文件合并过程中各个阶段消耗的时间，最终实验结果如图 14 所示.

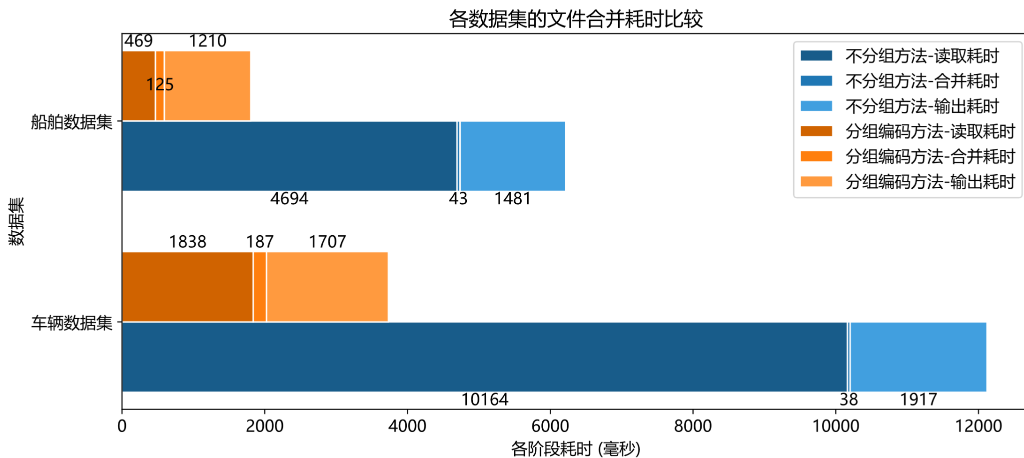


图 14 各数据集的文件合并耗时比较

在图 14 中，每个数据集分别使用了两种方法，因此有两个柱形图；而每个柱形图被分割成了三个部分，分别对应合并过程中的三个步骤消耗的时间. 这三个步骤分别是：读取碎片文件、在内存中合并与整理数据、将合并后的数据输出到新的 TsFile 中.

对比不分组方法和分组压缩方法，我们可以看到分组压缩方法所用的总时间更短，约为不分组方法的 30%. 观察各个阶段的耗时，可以发现耗时主要集中在文件读取与输入上. 一般来说，碎片文件的读取所花费的时间最多，合并结果的输出次之，用时最短的是在内存中的合并与整理. 在使用分组压缩方法后，文件读取的耗时明显缩短了，这得益于分组压缩方法将多个设备相同物理量名称的时间序列编码到同一数据块中，每次调用查询接口可以读取更多的数据，因此更快. 此外，在内存中合并整理数据所消耗的时间只占了全过程耗时的很小一部分，但仍可以看到分组压缩方法的耗时比不分组方法的耗时多，大约多消耗了 2~3 倍时间. 这主要是因为分组压缩方法需要对数据重新分组，并且重新拼接同一设备组中的相同物理量名称的时间序列，而不分组方法只需要将这些序列的多个碎片做合并即可. 因此，分组压缩方法在数据合并整理的阶段耗时自然会更长，但是总过程的耗时更短.

此外，在实际场景中，合并过程不会只进行一次，而是随着新增数据的到来而逐步合并的. 为了分析逐步合并过程中分组方法的空间优化情况，我们还分析了合并过程中文件大小的变化，结果展示在图 15、图 16 中，其中序列的平均长度与合并的碎片数量成正比. 换句话说，序列的平均长度越长，代表着合并了越多的碎片文件，图中横轴的变化则表示了碎片文件的逐步合并过程. 随着合并过程的进行，文件内序列的长度不断增加，短序列逐渐消失，变为长序列. 此时，不分组方法的存储效率也会逐步提升.

为了方便比较文件大小的变化，在图 15、图 16 中，我们将纵轴设置为归一化的文件大小，也即文件的绝对大小与其存储的数据点数的比值. 于是我们可以直观的看到存储同样多数据点时，序列的平均长度越长、文件内时间序列的数量越少时，不分组的 TsFile 文件大小越小，存储效率越高；而分组压缩方法存储的 TsFile 文件大小则相对稳定. 可以看到，随着碎片的逐步合并，每条序列的平均长度不断变长，对应的文件相对大小也在不断降低. 当合并了 100 个碎片文件后，不分组方法和分组方法的效率趋于一致. 此外，由于船舶数据集主要包含了字符串存储的船舶日志数据，采用字典编码的方法存储，因此它的长序列压缩比非常低，经过合并后文件大小会显著下降.

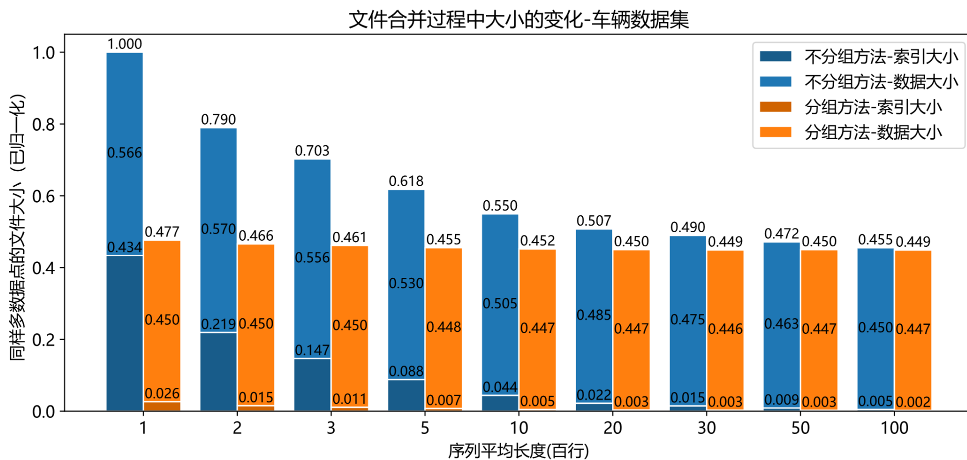


图 15　车辆数据集中碎片文件合并过程的大小变化

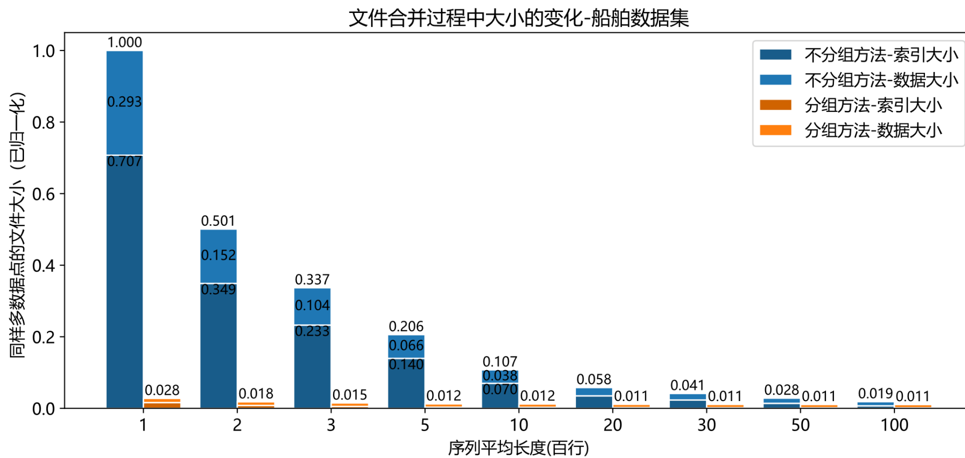


图 16 船舶数据集中碎片文件合并过程的大小变化

6 总结和展望

本文全面阐述了 Apache TsFile 中分组压缩方法的设计与实现. 针对短序列场景，我们深入分析了 TsFile 文件格式面临的挑战，挖掘出短时间序列出现的特征，并充分利用这些特性引出了分组压缩方法.

我们的分组压缩方法可以充分利用短序列场景下，设备元数据信息高度重复的特征，进行针对性的优化. 我们的方法解决了大量短序列压缩效率低、文件索引占比高、元数据冗余等挑战，优化了 Apache TsFile 文件格式的压缩比. 同时，我们通过实验进一步确定了压缩优化效果的原因，确定分组压缩方法在数据页压缩以及索引压缩方面都有所提升. 我们还通过实验验证了我们的方法兼顾了写入与读取性能，并且在碎片文件合并场景具有的优势. 随着短序列的合并，分组方法的优势逐渐减小，此时可以选用不分组方法进行 TsFile 存储，以保证查询性能.

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. 未来可以进一步推进端边云协同场景中端侧文件写入的优化工作，例如设计一种高效的流式写入与压缩方法提高效率. 未来可以研究一种可行的 TsFile 文件合并策略，来加速合并文件时数据整理的过程. 可以考虑支持用户自定义的预先分组的策略，例如根据用户常用的聚合查询来自动分组，提高数据管理与查询的效率. 此外，该分组方法通过拼接短时间序列来提高数据压缩效率，具有通用性，有进一步扩展到更多文件格式的潜力.

References:

[1] Wu D, Zhang P, Wang R. Smart Internet of things aided by “terminal-edge-cloud” cooperation[J]. Chinese Journal on Internet of Things, 2018,2(03):21-28 (in Chinese with English abstract).

[2] Qiao J. Research on File Structure Design andOptimization of Time Series DatabaseManagement System for Internet ofThings[D]. Tsinghua University, 2021.DOI:10.27266/d.cnki.gqhau.2021.000027 (in Chinese with English abstract).

[3] Wang C, Qiao J, Huang X, et al. Apache IoTDB: A time series database for IoT applications[J]. Proceedings of the ACM on Management of Data, 2023, 1(2): 1-27.

[4] 2024. https://iotdb.apache.org/

[5] Zhao X, Qiao J, Huang X, et al. Apache TsFile: An IoT-native Time Series File Format[J]. Proceedings of the VLDB Endowment, 2024, 17(12): 4064 - 4076.

[6] 2024. https://tsfile.apache.org/

[7] 2024. https://docs.influxdata.com/influxdb/clustered/

[8] 2015. https://www.influxdata.com/blog/new-storage-engine-time-structured-merge-tree/

[9] 2020. https://www.influxdata.com/blog/announcing-influxdb-iox/

[10] 2024. https://horaedb.apache.org/

[11] Xiao J, Huang Y, Hu C, et al. Time series data encoding for efficient storage: A comparative analysis in apache iotdb[J]. Proceedings of the VLDB Endowment, 2022, 15(10): 2148-2160.

[12] Xia T, Xiao J, Huang Y, et al. Time series data encoding in Apache IoTDB: comparative analysis and recommendation[J]. The VLDB Journal, 2024, 33(3): 727-752.

[13] Golomb S. Run-length encodings (corresp.)[J]. IEEE transactions on information theory, 1966, 12(3): 399-401.

[14] Blalock D, Madden S, Guttag J. Sprintz: Time series compression for the internet of things[J]. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2018, 2(3): 1-23.

[15] Welch T A. A technique for high-performance data compression[J]. Computer, 1984, 17(06): 8-19.

[16] Pelkonen T, Franklin S, Teller J, et al. Gorilla: A fast, scalable, in-memory time series database[J]. Proceedings of the VLDB Endowment, 2015, 8(12): 1816-1827.

[17] Liakos P, Papakonstantinopoulou K, Kotidis Y. Chimp: efficient lossless floating point compression for time series databases[J]. Proceedings of the VLDB Endowment, 2022, 15(11): 3058-3070.

[18] Fang C, Song S, Guan H, et al. Grouping time series for efficient columnar storage[J]. Proceedings of the ACM on Management of Data, 2023, 1(1): 1-26.

[19] Fang C, Chen Z, Song S, et al. On Reducing Space Amplification with Multi-Column Compaction in Apache IoTDB[J]. Proceedings of the VLDB Endowment, 2024, 17(11): 2974-2986.

[20] Bartík M, Ubik S, Kubalik P. LZ4 compression algorithm on FPGA[C]//2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS). IEEE, 2015: 179-182.

附中文参考文献:

[1] 吴大鹏,张普宁,王汝言.“端—边—云”协同的智慧物联网[J].物联网学报,2018,2(03):21-28.

[2] 乔嘉林.物联网时序数据库管理系统文件结构设计与优化研究[D].清华大学,2021.DOI:10.27266/d.cnki.gqhau.2021.000027.