软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(9):4285−4310 [doi: 10.13328/j.cnki.jos.007315] [CSTR: 32375.14.jos.007315]

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

基于神经网络的分布式追踪数据压缩和查询方法*

王 尚[1,2], 张晨曦[1,2], 彭 鑫[1,2]

[1](复旦大学 计算机科学技术学院, 上海 200438)

[2](上海市数据科学重点实验室 (复旦大学), 上海 200438)

通信作者: 张晨曦, E-mail: cxzhang20@fudan.edu.cn

摘 要: 分布式追踪数据作为一种重要的可观测性数据, 对性能分析、故障诊断、系统理解等运维任务起着至关重要的作用. 由于系统规模和复杂性的快速增加, 追踪数据的规模愈发庞大, 对存储提出了更高的要求. 为了降低追踪数据的存储成本, 数据压缩成为一种至关重要的方式. 现有的压缩方法无法充分利用追踪的数据特征实现高效压缩, 而且不支持对压缩数据的复杂查询. 提出了一种基于神经网络的分布式追踪数据压缩和查询方法. 该方法采用一种新的冗余抽取方式来识别追踪数据中的模式冗余和结构冗余, 并利用神经网络模型和算术编码实现高效的数据压缩. 同时, 该方法可以在压缩数据上进行高效查询, 而无需解压所有数据. 在 4 个开源微服务系统上收集多个不同大小的追踪数据集, 并对该方法展开评估. 实验结果表明, 该方法实现了较高的压缩比 (105.5–197.6), 平均是现有通用压缩算法的 4 倍. 此外, 还验证了该方法在压缩数据上的查询效率, 在最优情况下快于现有查询工具.

关键词: 分布式追踪; 无损压缩; 查询; 神经网络

中图法分类号: TP311

中文引用格式: 王尚, 张晨曦, 彭鑫. 基于神经网络的分布式追踪数据压缩和查询方法. 软件学报, 2025, 36(9): 4285–4310. http://www.jos.org.cn/1000-9825/7315.htm

英文引用格式: Wang S, Zhang CX, Peng X. Neural-network-based Compression and Query Approach for Distributed Tracing Data. Ruan Jian Xue Bao/Journal of Software, 2025, 36(9): 4285–4310 (in Chinese). http://www.jos.org.cn/1000-9825/7315.htm

Neural-network-based Compression and Query Approach for Distributed Tracing Data

WANG Shang[1,2], ZHANG Chen-Xi[1,2], PENG Xin[1,2]

[1](School of Computer Science, Fudan University, Shanghai 200438, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 200438, China)

Abstract: As an essential type of observability data, distributed tracing data plays a crucial role in operation and maintenance tasks like performance analysis, fault diagnosis, and system understanding. Due to the rapid increase in system scale and complexity, the volume of tracing data grows exponentially, putting forward higher storage requirements. To mitigate the storage cost of tracing data, data compression becomes a crucial approach. Existing compression methods fail to fully exploit tracing data features for achieving efficient compression, and they do not support complex queries on compressed data either. This study introduces a neural-network-based approach for compressing and querying distributed tracing data. It employs a novel redundancy extraction technique to identify pattern and structural redundancies within tracing data, and leverages neural network models and arithmetic coding to achieve efficient data compression. Meanwhile, the method enables efficient querying of compressed data without decompressing all the data. Variously sized tracing datasets are collected from four open-source microservices systems, and the proposed method is evaluated. Experimental results show relatively high compression ratios (105.5–197.6) are achieved by the proposed method, which are four times those of state-of-the-art general compression algorithms on average. Additionally, the querying efficiency of the proposed method on the compressed data is validated, showcasing faster performance than existing query tools in optimal scenarios.

Key words: distributed tracing; lossless compression; querying; neural network

* 收稿时间: 2024-01-15; 修改时间: 2024-06-06; 采用时间: 2024-10-30; jos 在线出版时间: 2025-06-04
CNKI 网络首发时间: 2025-06-05

在当今日益复杂和庞大的软件系统中, 分布式追踪作为一种关键的可观测性技术, 对保障系统的可靠稳定运行起着不可或缺的作用. 这项技术不仅使得我们能够跟踪和理解分布式系统中各个组件的交互行为[1], 更为重要的是, 它为系统性能分析[2,3]、故障排查[4–6]以及性能优化[7,8]提供了宝贵的数据支持. 追踪数据中所蕴含的信息, 例如系统调用、请求传递和资源利用等, 使我们可以深入洞察分布式系统在运行时的行为模式与性能瓶颈[9]. 除此之外, 分布式追踪提供了详细的系统运行时视图, 可以用于评估微服务系统的服务独立性和调用链复杂性[10], 对微服务架构的长期演进发挥了重要作用.

然而, 随着系统规模和复杂性的不断增加, 追踪数据量呈指数级增长, 对存储提出了巨大挑战. 以往的研究显示[11,12], 微服务架构通常包括数百到数千的微服务, 系统每天都会产生大量的数据. 例如, 腾讯的微信系统在超过 20 000 台机器上运行 3 000 多个服务, 每天的请求量通常在 10^{10}–10^{11}, 一天可以产生 PB 级别的追踪数据[13]. 此外, 许多追踪数据还需要长期存储, 通常根据开发周期以及运维需求需要保留几周或者几个月的时间. 历史追踪数据可以用于异常检测、性能分析以及系统优化. 然而, 存储和分析大量追踪数据会带来巨大的成本. 尽管难以获取公司具体的存储成本, 但有研究估计, 运营成本的下限可能是每月每 GB 约 2 美分[14]. 这意味着对于像腾讯、eBay 这样的公司来说, 每年需要花费上百万美元存储生成的追踪数据. 因此, 存储追踪数据不仅有保障分布式系统持续提供稳定可靠服务的重要性, 而且也具有数量庞大成本高昂的挑战性.

目前有两种主流方法用于减少追踪数据的存储开销. 一种是通过采样的方式减少追踪数据生成和收集的数量, 另一种则是通过压缩减少追踪文件的大小. 常见的采样方法包括随机采样和均匀采样, 此外还有许多研究[11,12,15–17]提出了不同的算法用于采样追踪数据. 虽然通过采样可以有效地减少需要存储的追踪数据, 但是采样后的数据规模依旧不容小觑. 因此, 在将追踪数据归档之前, 有必要通过数据压缩来降低存储成本. 现有的分布式追踪工具通常直接使用各种数据库 (如 Elasticsearch、ClickHouse、S3 等) 进行追踪数据的存储. 当前, 在实践中主要使用传统的文本压缩算法对分布式追踪数据进行压缩, 如 gzip[18]采用 LZ77 算法[19]和霍夫曼编码[20]等方法. 尽管这些方法可以快速捕获数据中的局部冗余, 但通常不是最优的, 且并未针对追踪进行优化, 因此在压缩追踪数据时效果较差. 另一方面, 现有的压缩算法在设计时并未充分考虑对压缩数据的搜索问题. 对这些方法压缩后的数据进行搜索是相对耗时的, 需要依次解压所有数据, 然后再筛选出满足条件的内容. 为了提高查询的响应速度, 现有的追踪查询工具 (如 Grafana Tempo[21]和 Jaeger[22]) 通常将保存的数据分块并建立外部索引. 在查询时, 这些工具搜索对应的索引, 只解压可能包含与查询条件匹配的数据块. 然而, 这种方法以大量的存储空间和内存为代价, 传入的追踪文件越大, 建立索引所消耗的空间也越多.

在追踪数据中主要存在模式冗余和结构冗余两种形式的冗余. 模式冗余是指追踪数据的每个跨度中存在的键值对冗余, 即不同跨度间重复的字段和取值组合. 例如: 对于调用相同服务实例产生的不同跨度, 其中表示服务实例、服务名的键值对在这些跨度中都是相同的, 而这些相同的键值对会随不同跨度被系统重复存储, 从而产生大量冗余. 结构冗余是指不同的调用链中存在的相同调用结构. 在分布式追踪中调用链通常可以表示为由跨度组成的有向无环图, 该图结构反映了系统处理请求时具体的服务调用逻辑. 这时, 对相同功能的请求所产生的调用链就可能呈现出同样的调用结构, 如不同用户登录产生的调用链结构可能相同. 另一方面, 在一个微服务系统中一些服务提供的功能可能会被不同请求使用, 例如鉴权等基础功能可能会被多种请求频繁使用, 这时可能会导致不同调用链中存在部分调用结构相同. 这些重复的调用结构也会导致数据在存储时产生大量的冗余信息.

针对上述问题, 本文提出了一种基于神经网络的分布式追踪数据的压缩和查询方法 NCQT (neural-network-based compression and query method for distributed tracing data). NCQT 通过识别追踪中的模式冗余和结构冗余, 减少数据中的重复内容, 并训练了一个基于神经网络的编码器进行数据压缩, NCQT 允许对压缩数据进行高效的搜索, 而无需解压所有追踪数据. 首先, NCQT 创建了多个不同的索引字典以记录追踪数据中的模式冗余和结构冗余, 并将自然语言文本转换为格式规整的数字表示, 以简化和加快编码器的学习过程. 然后, NCQT 将处理后的数据分组, 并利用训练过的编码器和算术编码[23]创建追踪数据的最小二进制表示. 在查询和解压阶段, NCQT 可以利用索引字典和编码器完整恢复追踪数据的文本表示.

为了评估 NCQT 的有效性和效率, 我们在 4 个具有较高影响力的开源微服务系统上收集了不同大小的追踪

数据集, 并在其上展开了一系列的实验研究. 结果显示, NCQT 达到了较好的压缩效果, 与现有的压缩方法相比, NCQT 在压缩比上提升了 2.4–9.9 倍. 其次, NCQT 的压缩效率高于基于神经网络的压缩算法, 其压缩速度相较于对比方法提高了 41.2–161.7 倍. 此外, NCQT 可以在压缩数据上实现较为高效的查询, 在最优情况下能快于目前主流的追踪查询工具.

综上所述, 本文的主要贡献如下.

● 提出了一种通用的分布式追踪冗余抽取方法, 可以减少追踪数据中的模式冗余和结构冗余.

● 提出并实现了一种基于神经网络的分布式追踪压缩和搜索方法, 利用神经网络模型进一步捕获追踪数据中的冗余, 并可以在压缩数据上进行有效的查询.

● 设计了一系列实验验证了方法的有效性和效率, 并研究了一些重要参数配置的影响.

本文第 1 节介绍压缩和查询的相关工作. 第 2 节介绍本文所需的基础知识, 包括分布式追踪背景知识和算术编码. 第 3 节介绍本文提出的基于神经网络的分布式追踪压缩和查询方法. 第 4 节通过多组实验验证了所提方法的有效性和效率. 第 5 节对本文内容进行总结, 并展望未来的工作.

1 相关工作

传统的文本压缩算法通常可以分为 3 类: 基于熵、基于字典以及基于预测. 基于熵的压缩方法首先收集文本文件的统计信息, 然后为每一个字符设计变长编码. 例如, 霍夫曼编码[20]会为频繁出现的符号分配短编码, 而算术编码[23]则为每个符号分配一个区间, 并根据输入持续更新数据的编码区间. 这些方法具有单一字符的编码机制, 而统计模型的构建方式决定了压缩的质量. 基于字典的压缩方法在压缩过程中搜索类似的标记, 并在处理输入流时将他们存储在字典中. LZ77[19]算法使用三元组维护字典, 分别表示偏移量、重复长度以及偏差字符, 在压缩时用三元组替代重复出现的子序列, 并根据滑动窗口动态更新字典. LZ77 的变体 LZMA[24]和 DEFLATE[25]是常用压缩工具 lzma 和 gzip 的算法基础. 基于字典的方法通常假设输入文本中出现的模式紧密出现, 因此难以消除长距离的冗余. 基于预测的压缩方法在读取输入流时基于当前上下文预测下一个字符, 并在预测成功时分配更短的编码. PPM[26]是该类算法的典型代表, 它利用几个固定顺序上下文模型预测输入序列的下一个字符, 并结合编码器生成文本的最终表示. 基于预测的方法的压缩效果取决于方法的预测准确率, 变量的出现会降低预测精度.

神经网络在预测任务上表现出色, 因此基于神经网络的压缩方法受到了人们的广泛关注[27–33]. 研究表明, 与传统的有限上下文模型相比, 神经网络模型通常可以更好地学习复杂的数据模式, 从而降低预测误差[27], 预测精度越高, 算术编码的压缩效果越好. Schmidhuber 等人[28]采用三层前馈神经网络替换 PPM 预测器的上下文匹配方法, 证明了神经网络模型在无损压缩上的潜力. Mahoney 等人[29]使用两层前馈神经网络, 并通过单次学习以预测下一个字符, 使得神经网络压缩更加高效. 然而这两种方法都使用了前馈神经网络, 仅考虑了字符的前一个窗口, 无法捕获长期依赖. Liu 等人[30]使用三层 LSTM 作为模型结构, 为了捕获更多上下文信息以进行更好的预测, 该方法引入了循环连接以保留结束时的隐藏状态, 并将其复用为下一批的初始状态. Goyal 等人[27]基于 biGRU 实现了一个自适应和半自适应训练的混合架构, 该方法无需额外的训练数据集, 且不限于特定的数据类型. Mao 等人[33]提出了一种基于单层 Transformer 的无损压缩器, 方法通过字节分组和共享前馈神经网络的方式充分利用单层 Transformer 的容量, 实现了更高的压缩比和效率. 然而, 现有的基于神经网络的压缩方法主要关注压缩比, 而牺牲了压缩速度, 且大多数模型需要充分学习压缩数据的特征以达到较高的预测精度, 进一步增加了压缩的时间开销.

近年来, 许多研究提出了针对日志的压缩方法[34–41], 对同为运维数据的追踪数据压缩具有重要的指导意义. 现有的日志压缩方法通常可以分为日志变换和文本替换两类[35]. 日志变换主要通过变换现有的日志以提高文件压缩效果. Christensen 等人[36]利用聚类算法将每行日志分配到不同的桶中, 并以桶为单位进行压缩. Lin 等人[37]将日志条目按列分割, 并利用模型压缩每列, 在解压效率上优于现有方法. 文本替换的核心思想是用较短的表示替换日志中的冗余内容. Liu 等人[38]利用快速迭代聚类, 将日志数据拆分为由模板组成的常量部分以及由参数组成的变量部分, 进而生成适用于通用压缩算法的中间表示. Wei 等人[34]在模板提取的基础上增加了增量时间戳、相关识别以及弹性编码这 3 种处理方法, 实现了更高的压缩性能. Rodrigues 等人[39]使用一组用户指定的匹配变量的规

则来解析日志, 并将剩余的部分作为日志的模板. 不同于上述的压缩方法, Ding 等人[40]将日志冗余提取方法和基于神经网络的压缩方法相结合, 建立了一个高效的日志存储系统. Li 等人[41]通过最长公共子序列和基于熵的方法识别日志数据中的共性和可变性, 并基于此设计了多种编码规则, 从而用更短的形式表示日志数据.

目前, 对压缩数据进行查询的方法[39,42−45]主要分为两种: 直接在压缩的数据上查询而无需解压, 以及划分数据并在查询时过滤无关单元. 第 1 种方式通常需要对数据建立索引, 这将带来额外的存储开销. Agarwal 等人[42]提出了一种熵压缩的方法, 将部分信息嵌入到压缩数据中, 以减少索引的存储大小. Pibiri 等人[43]将基于字典的技术应用于倒排索引的压缩, 以实现压缩比和压缩效率之间的平衡. Zhang 等人[44]基于 Sequitur 实现了一种分层压缩算法, 可以避免不必要的重复处理. 第二种方式可以采用高压缩比的压缩方法, 但查询效率会略微降低. Rodrigues 等人[39]通过将日志处理为格式整齐的模板和参数, 实现了对压缩数据的快速查询. Wei 等人[45]则将日志参数进一步细分为更小的单元, 能够更加精细地筛选出符合要求的日志, 减少解压数据量.

现有的方法主要是针对普通文本和日志数据进行设计的. 而分布式追踪数据由跨度组成, 且跨度可以构成反映请求在系统中执行过程的调用链, 这与普通文本数据和日志数据都存在较大的区别. 同时, 在对分布式追踪数据进行压缩时, 需要考虑分布式追踪数据中的模式冗余 (不同跨度间重复的键值对) 和结构冗余 (不同调用链间重复的调用结构), 这样才能实现分布式追踪数据的高效压缩. 因此, 本文提出了基于神经网络的分布式追踪数据压缩和查询方法, 通过识别分布式追踪数据中的模式冗余和结构冗余, 实现了对追踪数据的高效压缩, 并且能够在不引入额外存储开销的情况下实现对已压缩追踪数据的复杂查询.

2 基础知识

本文针对分布式追踪数据提出了一种压缩和查询的方法, 下面就相关概念和基本知识予以介绍.

2.1 分布式追踪背景知识

分布式追踪记录了请求在多服务架构中传播所采用的路径, 是微服务基础设施的重要组成部分. 目前, 针对微服务的分布式追踪已经有多种实现方式, 例如 Zipkin、Jaeger 和 OpenTelemetry 等. 虽然不同工具产生的追踪数据在格式上不完全相同, 但在数据模型上基本上都由追踪、跨度和跨度上下文这 3 个核心概念组成. 追踪表示一次请求的追踪过程, 代表了一次请求的完整请求链路, 使用唯一的追踪标识符标识. 跨度是调用链中的基本组成, 表示系统中具有开始时间和执行时长的逻辑单元. 每个跨度都封装了多种状态, 包括时间戳、关联事件以及标签信息等. 跨度上下文涵盖了追踪中标识跨度的所有信息, 且随请求传播到子跨度, 是支撑分布式追踪的关键. 跨度上下文包含了从父跨度传播到子跨度的追踪标识符、跨度标识符和选项等, 子跨度利用这些信息构建服务间的调用关系. 如图 1 所示, 追踪由树形结构的一组跨度组成, 每个跨度对应一个服务调用. 除了根跨度以外, 每个跨度都有一个父跨度, 它可以开启当前跨度. 例如, 在图 1 中, 跨度 A 是一个同步调用, 在执行过程中分别开启了跨度 B 和跨度 E 两个子跨度, 因此跨度 A 是跨度 B 和跨度 E 的父跨度. 跨度上下文信息随着服务调用从跨度 A 向下传播, 用于在子跨度中生成追踪 ID 和父跨度 ID.

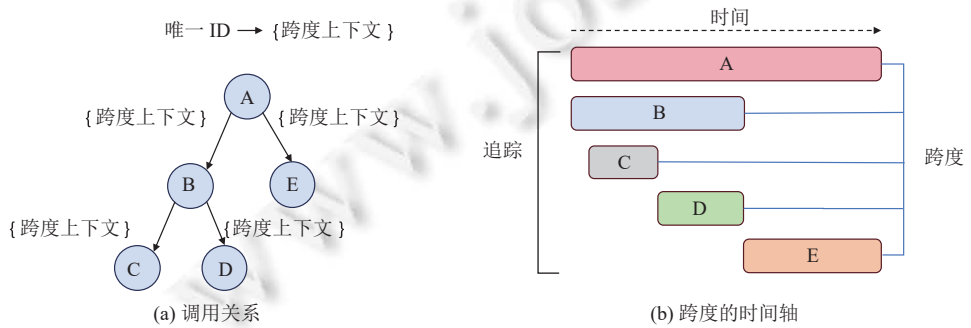

图 1 分布式追踪的调用链

图 1(b) 显示了图 1(a) 调用关系下跨度的时间轴. 当服务接收到请求时, 执行相应的操作并生成跨度, 同时记录服务调用的开始时间. 如果需要调用下游服务, 则将跨度上下文随请求传递到下一个服务. 下游服务收到请求后提取跨度上下文, 执行相关操作并利用跨度上下文生成自己的跨度信息, 重复以上过程直至子跨度不再调用其他服务. 当服务调用返回时, 记录结束时的时间戳并关闭此跨度.

2.2 分布式追踪数据结构

自从 Google 在 Dapper 中[46]提出分布式追踪的概念以后, 不同的分布式追踪技术基于此提出了不同的分布式追踪标准. 例如, CNCF 提出的 OpenTracing 制定了一套无关厂商、无关平台的协议标准, Zipkin 和 Jaeger 均遵循 OpenTracing 协议; 而 Google 开源的 OpenCensus 不仅制定了分布式追踪的规范, 而且还提供了不同开发语言的实现以及连接协议. 为了汇集不同分布式追踪协议的优势, CNCF 将 OpenTracing 和 OpenCensus 合并为 OpenTelemetry, 其逐渐成为追踪领域新的国际化标准.

图 2 展示了符合 OpenTelemetry 规范的追踪数据结构. 每条追踪数据对应于一个资源跨度列表 (resourceSpans), 列表中的每个元素都代表一组资源 (resource) 下的跨度. 资源捕获了与记录追踪数据的实体相关的信息. 以部署在 Kubernetes 容器中的服务产生的追踪数据为例, 资源记录了对应节点的名称、命名空间、以及 Pod 和容器的名称等相关信息. 作用域跨度 (scopeSpans) 包含了一组具有相同资源的跨度, 每个跨度可能属于不同的作用域 (scope). 作用域对应于应用程序代码中的逻辑单元, 开发人员可以选择适当的检测范围. 跨度 (spans) 包含了一组具有相同作用域的跨度, 其中每个跨度代表追踪中的一个操作或事件. 跨度数据中通常包含了跨度上下文信息 (traceId 和 spanId)、操作名、开始和结束的时间戳以及由键值对组成的属性列表等信息. 除了根跨度以外, 每个跨度都记录了一个指向父跨度的标识符 (parentSpanId).

```
{
  "resourceSpans": [{
    "resource": {
      "attributes": [{
        "key": "service.name",
        "value": {
          "stringValue": "hello"
        }
      }]
    },
    "scopeSpans": [{
      "scope": {
          "name": "io.opentelemetry.tomcat",
          "version": "1.0.0"
      },
      "spans": [
      {
          "traceId": "7bba9f33312b3dbb8b2c2c62bb7abe2d",
          "spanId": "086e83747d0e381e",
          "parentSpanId": "051581bf3cb55c13",
          "name": "hello-api",
          "startTimeUnixNano": 1709969302000000000,
          "endTimeUnixNano": 1709970000000000000,
          "kind": "SPAN_KIND_SERVER",
          "attributes": [
          {
            "key": "http.request.method",
            "value": {
              "stringValue": "GET"
            }
        }]
      }]
    }]
  }]
}
```

图 2 符合 OpenTelemetry 规范的追踪数据示例

3 基于神经网络的追踪压缩和查询方法

为了高效存储和查询分布式追踪数据, 本文提出了一种基于神经网络的追踪压缩方法 NCQT. 方法利用神经网络强大的上下文学习能力进一步提高追踪的压缩比, 并通过预处理和分组解决神经网络压缩和解压效率低下的问题. 图 3 显示了 NCQT 压缩和搜索架构的整体流程图, 主要可以分为 3 个阶段: 追踪数据冗余抽取、基于神经网络的编码器获取和数据压缩, 以及追踪数据搜索和解压. 其中, 追踪数据冗余抽取是在线的, 而编码器获取和数据压缩可以在归档时离线进行.

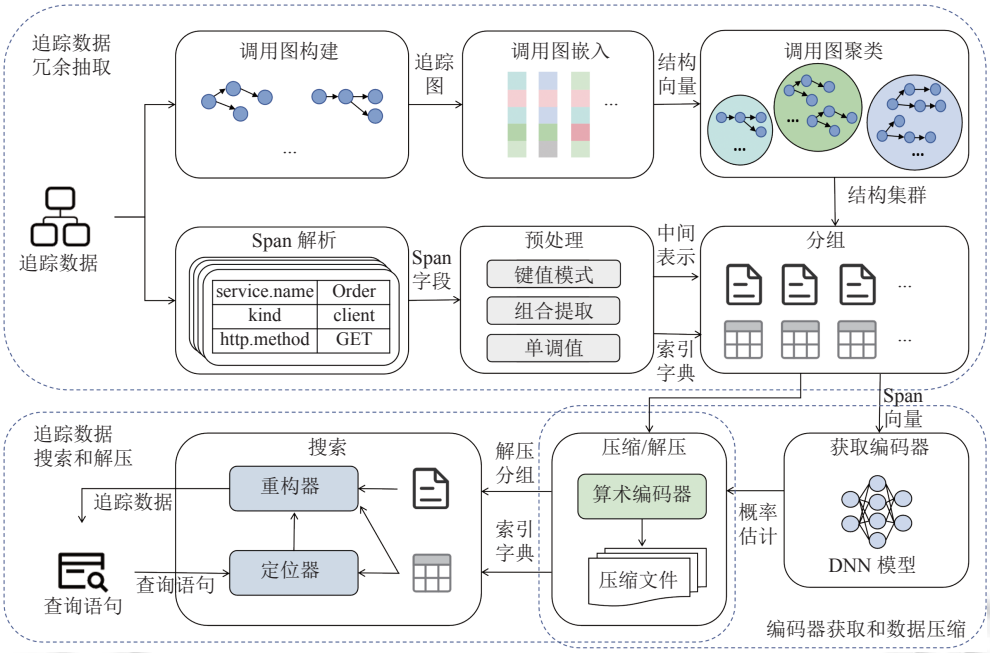

图 3 NCQT 流程图

● 追踪数据冗余抽取. 这一阶段主要利用追踪数据的特征进行初步的冗余约减, 并将数据组织为易于查询的中间表示. 在处理来自分布式系统的追踪数据时, NCQT 首先将其解析为由跨度组成的数据表示以及由调用关系组成的结构表示. 对于跨度数据, NCQT 将解析后的数据交由对应的跨度容器进行预处理. 对于结构数据, NCQT 通过识别不同结构之间的相似性将其划分为不同的集群. 预处理后的追踪数据会按照结构集群分组, 作为压缩和解压的最小单元.

● 编码器获取和数据压缩. 这一阶段利用神经网络将数据压缩为更短的二进制表示. NCQT 首先对分组数据进行采样, 利用少量的数据训练神经网络模型, 进而获取一个学习了数据特征和表示方式的编码器. 在压缩追踪数据时, NCQT 利用训练过的编码器并行处理每个分组的数据, 获取上下文向量的概率估计, 并结合算术编码生成更短的二进制表示, 最终为每个分组生成一份压缩文件.

● 追踪数据搜索和解压. NCQT 允许用户使用指定的查询语句对追踪数据进行查询. 在搜索阶段, NCQT 解析用户输入的查询语句, 并判断压缩数据中是否存在满足要求的数据. 然后, NCQT 查找所有包含了目标追踪数据的分组, 并将其交由神经网络解压. 最后, NCQT 筛选出分组中满足条件的追踪数据, 还原数据的文本表示并返回最终的结果.

3.1 追踪数据冗余抽取

在获取编码器并压缩数据之前, NCQT 首先利用追踪数据的特征消除数据中的模式冗余和结构冗余, 并将其重新组织为易于查询和压缩的数据格式. 追踪数据由服务调用过程中生成的跨度数据组成, 每条跨度数据记录了

该跨度自身的属性字段和父跨度生成的上下文字段. 因此, 一条追踪数据可以视为由所含跨度的字段信息以及跨度之间的调用关系两部分组成. NCQT 在解析追踪文件时, 首先将跨度数据规范化为统一的格式, 并从中提取出追踪的结构图.

如图 4 所示, 以符合 OpenTelemetry 规范的追踪数据为例, 数据包括此消息中跨度的资源以及源自该资源的跨度列表. 每个跨度是由服务调用中的一次操作生成的, 代表应用程序的一次处理过程. 因此, 对于同一操作产生的跨度数据, 其包含的属性字段相对固定, 受限于该操作的业务范围. 例如, 对于数据库查询的跨度“mongodb.find”, 该数据中包含的属性字段通常只与数据库相关, 而对于解析主机名的跨度“dns.lookup”, 其属性字段往往只包含 IP 相关的内容. 因此, NCQT 将一条追踪数据中的跨度处理为统一的键值表示, 按照操作种类交由不同的跨度容器进行并行处理. 这种方式不仅有利于将相似的数据聚集在同一容器以便于更好地冗余提取, 也有利于搜索时更快速地判断该种类的跨度是否包含满足查询条件的数据. 同时, NCQT 从上下文字段中提取跨度的调用关系, 并交由结构容器构建追踪的树状结构图, 以供后续通过结构相似性对追踪数据重新排序.

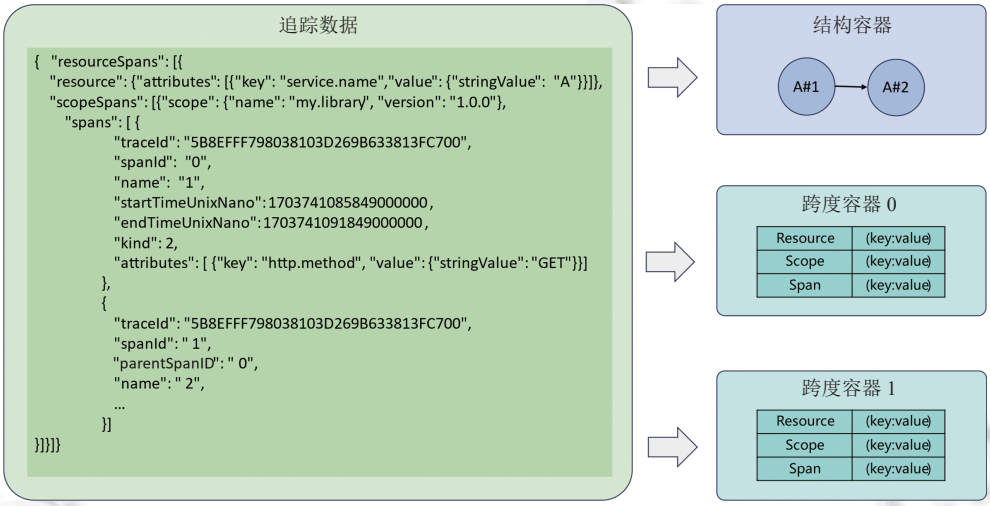

图 4 追踪数据解析

3.1.1 预处理

追踪数据中存在着大量重复记录的字段和值, 预处理阶段的主要目的是删除跨度中高度重复的部分, 并以特殊格式对它们进行编码, 为后续应用神经网络压缩以及搜索时快速定位做好准备. 具体而言, NCQT 应用了 3 条预处理规则: 键值模式替换、组合提取和单调值处理.

● 键值模式替换. 此规则的思路在于用短的数字替换长的键值字符串表示. 每个跨度是由键值表示的字段组成, 尽管服务可以自定义字段, 但字段的总数有限, 且大部分字段拥有相对确定的取值范围. 例如, 在第 4 节实验所用的 robot-shop 数据集中, 尽管有上百万个跨度, 但是字段种类仅有 91 个, 且多数字段的取值种类少于 100, 仅有部分数值类型的字段取值相对较多. NCQT 以跨度为单位将所有的字段键值对转换为数字表示. 例如, 跨度条目 (kind:client, http.status_code:200) 在键值模式替换后会被转换为 (0:0, 1:0). 同时, NCQT 在跨度容器中记录包含映射关系的索引字典, 以便后续将数字还原回其原始格式.

● 组合提取. 此规则侧重于避免重复记录相同的字段名. 对于由同一个操作产生的跨度数据, 其字段通常是由有限范围的属性组合生成, 因而每个跨度中都记录了相似的字段列表. NCQT 使用组合字典记录了每种跨度所有可能的字段组合, 处理后的跨度数据仅需要记录对应组合的 ID, 并按照组合的顺序依次记录参数的数值索引. 例如, 通过组合映射关系 (0)→(0, 1), 可以将规则一生成的键值对组合 (0:0, 1:0) 进一步转换为 0:(0, 0). 通过这种方式不仅可以将数据编码为更简短的形式, 而且有利于在查询数据时快速定位目标字段在编码数据中的位置.

● 单调值处理. 在跨度数据中, 通常可以看到一些字段具有单调的值, 尤其对于连续的条目, 可能会有相近的取值, 这些字段包括时间戳、数据库事务标识符、计数器等. 例如, “startTimeUnixNano”和“endTimeUnixNano”字段是由 19 个字符的长字符串组成, 而一个请求通常在秒甚至毫秒级别即可执行完成, 时间戳中代表年份、月份和日期部分的数字是多余的. 对于这些单调值的字段, NCQT 首先记录最小的值, 然后用偏移量代替原始值, 从而删除大量不必要的数字. 在解压缩时, 只需要通过最小值和偏移量重新计算实际值. 在图 4 中可知, 跨度大多数字段涵盖了字段取值的类型, 因此很容易即可找出可以处理的字段, 而对于跨度开始和结束时间, 我们记录根跨度的开始时间, 对于每条追踪以此作为基准值计算每个跨度的时间偏移.

图 5 展示了一个跨度容器如何抽取数据冗余并返回编码后的数据. 最终的输出为一个元组, 其中包括一个 32 位组合 ID, 两个 64 位的时间偏移以及一个 32 位参数 ID 组成的序列, 序列的长度对应于该组合下字段的个数. 这种结构旨在提供对跨度数据的紧凑表示, 有助于进一步的压缩和处理, 同时确保了重要信息的完整性.

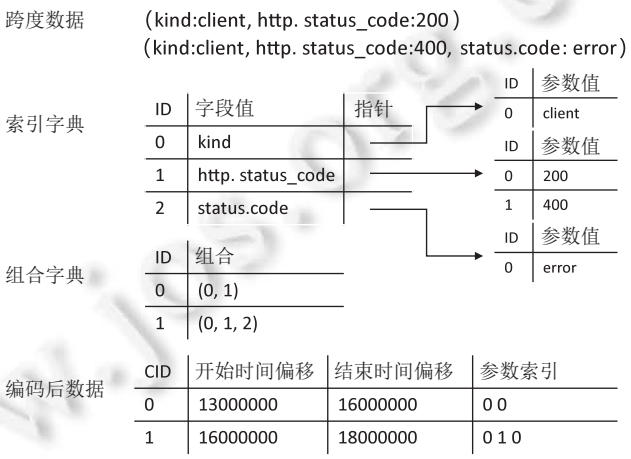

图 5 预处理

3.1.2 调用图嵌入和聚类

在原始的追踪文件中, 追踪按照收集的时间顺序存储, 不同请求产生的追踪交错排列. 这种排列方式不利于压缩算法识别数据中的重复模式, 且进一步增大了搜索的时间成本. 图嵌入和聚类旨在通过识别不同追踪结构的相似性, 将具有相似跨度组成或调用关系的追踪数据组织在一起, 减少局部信息熵, 进而提高编码器的压缩率以及搜索器的查询效率.

追踪的调用图是由跨度节点组成的多叉树, 理论上, 基于树编辑距离的方法可以更好地衡量追踪结构之间的差异. 然而, 在许多微服务系统中它们的成本太高, 因为比较所需的时间复杂度为 $O(N^2)$, 其中, N 为追踪调用图中包含的跨度数量. 因此, 已有研究提出了一些轻量级的嵌入方法[6,17]以获取追踪图的向量表示, 从而减少不同追踪间比较的时间开销. 然而, 这些方法要么忽略了追踪的树状结构, 要么与具体的跨度数据相关. 因此, 我们提出了一种轻量的基于 N-gram 的追踪结构嵌入方法, 通过在图上滑动一个长度为 N 的窗口提取局部特征.

如算法 1 和图 6 所示, NCQT 通过以下方式生成追踪的表示. 首先, 通过深度优先搜索的顺序遍历每一个调用图的节点, 并将节点加入从根节点开始的路径列表, 其中每一层中优先遍历开始时间较早的节点. 当列表长度不小于 N 时, 将列表末尾 N 个节点的服务调用序列作为尾节点的调用路径. 如果遍历到叶子节点时列表长度小于 N, 则将从根节点开始的完整调用链加入该调用图的路径表中. 最后, 方法收集所有调用图的调用路径, 并根据调用路径为每种结构生成一个向量表示. 具体而言, 方法判断每个调用路径的存在性, 如果某个调用路径存在, 方法将该维度的取值设为此调用路径的数量, 否则设为 0. 值得注意的是, 由于一个追踪中可能含有重复的跨度, 因此可能存在某个调用路径出现多次的情况.

算法 1. 嵌入过程.

输入: 追踪图的根节点列表 *roots*, 窗口长度 *N*;
输出: 追踪图的向量表示 *embeddings*.

```
function get_embeddings(roots, N)
    embeddings ← [], paths ← []
    for each root in roots
        paths ← get_paths(root, N) ∪ paths //获取所有结构的路径列表
    vocab ← set(paths) // 将路径的集合作为词汇表
    for each path in paths
        counter ← Counter(path), embedding ← [] // 初始化每个结构的路径计数器和向量表示
        for each word in vocab
            embedding ← counter.get(word) ∪ embedding //每个维度取值为路径数量
        embeddings ← embedding ∪ embeddings //将该结构的向量表示加入结果列表
    return embeddings
function get_paths(root, N)
    all_paths ← [], path ← stack()
    function dfs(node, depth)
        path.push(node)
        if depth ⩾ N
            all_paths ← path[–N:] ∪ all_paths //取末尾 N 个节点的序列作为尾节点的调用路径
        if depth < N and node is leaf
            all_paths ← path ∪ all_paths //叶子节点若路径长度不足 N, 则将整个序列作为其调用路径
        for each child in node.children
            dfs(child, depth + 1)
        path.pop()
    dfs(root, 1) //从树的根节点开始向下搜索
    return all_paths
```

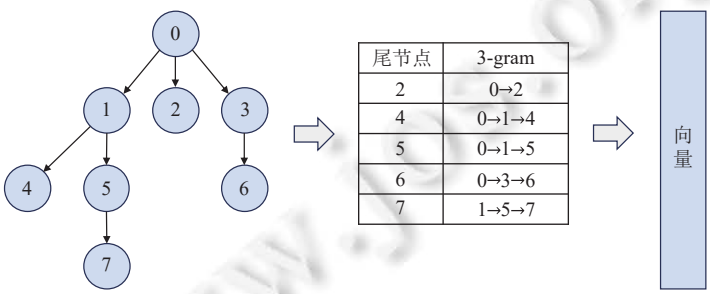

图 6 图嵌入

在聚类算法上, NCQT 使用 K-means 算法聚类所有种类的调用图. 对调用图聚类旨在为后续分组提供依据, 进一步提高压缩的效果和查询的效率. 由于聚类主要是为后续的分组压缩和搜索提供辅助支持, 即使聚类效果欠佳, 方法依然能够实现对追踪数据的无损压缩, 但复杂的聚类算法会带来更多的压缩时间开销. 为了平衡聚类速度和聚类效果, 我们选择线性时间复杂度的聚类算法 K-means. K-means 算法[47]是一种广泛使用的聚类方法, 可以快速

达到局部最优效果且具有较强的可解释性, 并且方法的参数较少. 为了确定最优的 k 值 (聚类数), 我们使用轮廓系数 Silhouette Coefficient 以评估不同 k 值下的聚类性能. 轮廓系数综合考虑聚类的密集程度及分离程度, 为每个样本提供一个度量, 其计算公式如下:

$$s(i)=\frac{b(i)-a(i)}{\max\{a(i),\ b(i)\}} \tag{1}$$

其中, $a(i)$ 代表第 i 个样本与同一聚类簇下其他样本的平均距离, $b(i)$ 代表第 i 个样本与最近邻聚类簇中样本的平均距离, 整个聚类结果的轮廓系数为所有样本轮廓系数的平均值. 度量的范围是−1 至 1, 数值越大说明样本点所在簇与相邻簇之间的距离越远, 聚类的效果越好. 由于轮廓系数只有在标签数大于 1 且小于样本数时有效, 因此对于 S 种结构, NCQT 设置 k 在范围 [2, S−1] 内依次进行 K-means 聚类, 并计算每次聚类的轮廓系数, 选取分数最高的作为最终的聚类结果.

3.1.3 分 组

分组的主要目的是提高神经网络压缩的效果和效率. 在同等规模的数据上, 基于神经网络的压缩方法相较于传统压缩方法带来了更大的时间开销. 通过分组将原本大文件分割为多个更小的文件, 可以实现并行数据压缩, 进而提高压缩的效率. 此外, 为了平衡压缩时间和效果, 基于神经网络的压缩方法大多采用单层或少数几层的网络结构, 当追踪的种类较多时, 神经网络难以充分学习不同追踪的数据特征, 导致预测准度下降. 基于前文聚类的结果, 将结构相似的追踪数据聚集在一个数据块中作为压缩和解压的最小单元, 有助于神经网络更好地学习和预测局部数据特征.

分组的另一个目的是提高查询的效率. 现有的压缩方法在解压时通常需要按照编码时的顺序依次解码数据, 而将数据进行分组赋予了从不同位置开始解压数据的灵活性. 在搜索阶段, 只有包含满足查询条件的追踪数据的分组需要被解压, 有效地降低了搜索过程中所需解压的数据量. 同时, 利用聚类结果进行分组可以将具有相似特征的追踪数据聚集在一起, 提高了目标数据在相同或相邻分组的概率, 进而减少了需要解压的分组数量.

分组方法优先聚集拥有相同结构的追踪数据, 其次将根据聚类簇将结构相似的追踪数据分在同一组或相邻组. 具体而言, NCQT 首先将分组大小设置为每组 L 条追踪, 然后按照聚类结果对所有结构进行排序, 最后按照排序结果重新组织追踪数据, 每 L 条数据切分为一个分组. 分组完成后, NCQT 将每个分组所含追踪的 ID 列表、结构列表以及开始时间列表记录为该分组的索引字典, 以便搜索时快速判断分组中是否存在满足查询条件的数据, 并定位其在分组中的位置.

3.2 编码器获取和数据压缩

NCQT 采用半自适应性神经网络压缩方法. 首先, 模型利用部分输入序列进行训练, 训练后的模型将作为压缩阶段的初始模型. 在压缩过程中, 模型动态更新参数, 并利用所产生的概率估计作为算术编码的输入, 用于生成压缩文件. 在前文的预处理阶段, 跨度数据已经从自然语言文本转换为由数字组成的列表, 我们使用特殊数字作为跨度的分隔符, 将追踪数据编码为取值范围在 [0, 22] 之间的一维列表. 这种设计使得模型能够更加高效地处理追踪数据, 并在压缩时提供更好的性能.

3.2.1 编码器

理论上, 所有可以处理序列数据的网络结构都可以作为用于文本压缩的模型. 在本文的实现中, 我们参考 TRACE [33]和 DZip[27]的工作, 选择了 Transformer 和 GRU 两种网络结构作为可选的编码器. 图 7(a) 显示了以 Transformer 作为编码器时的网络设计. 传统的 Transformer 通常由多层编码器和解码器组成, 用于提取自然语言序列中的高级语义特征[48]. 然而我们实践发现, 只需要单层编码器进行压缩就可以获得较高预测准确率, 而且并没有引入更多的时间成本. NCQT-Transformer 的网络结构包括单层 Transformer, 一个全连接层, 以及最后用于输出预测概率的 SoftMax 层. 其中, Transformer 层由多头注意力以及全连接的前馈神经网络组成. 图 7(b) 则展示了以 GRU 作为编码器时的网络结构, 模型包括两个双向门控循环单元、多个线性层以及最后的 SoftMax 层. 模型的输入序列经过双向门控循环单元后首先被堆叠并扁平化为一个向量, 然后分别输入到两个线性层中, 其中左侧带

ReLU 的线性层主要用于学习输入中的长依赖关系. 模型将两个线性层的输出拼接在一起, 然后经过 SoftMax 层生成下一个字符的概率估计.

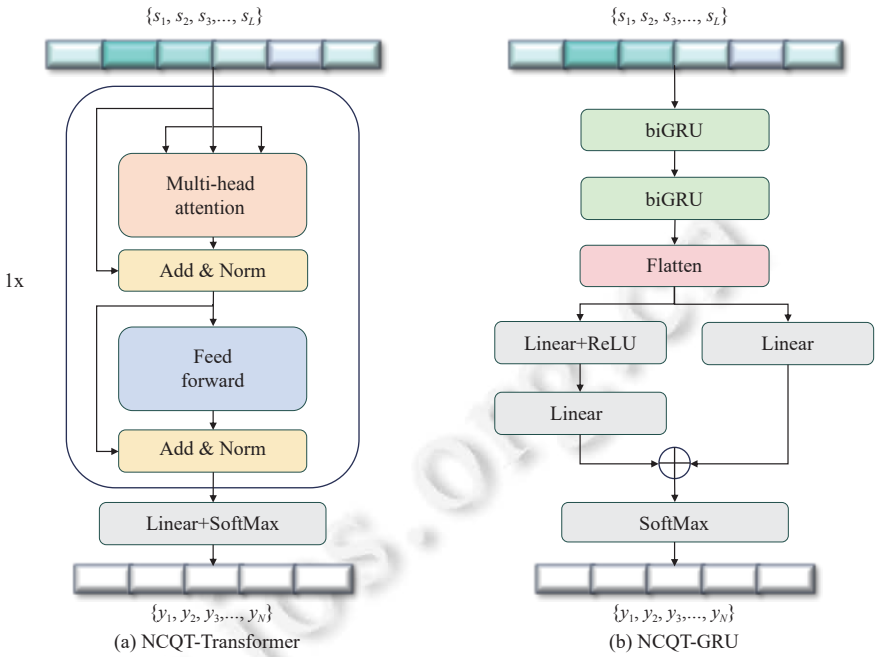

图 7 NCQT 中神经网络的模型结构

模型以一个长度为 L 的序列 $S=\{S_1, S_2, S_3, \ldots, S_L\}$作为输入, 并预测该序列的下一个字符 S_{L+1}, 其中, L 为预测窗口的大小. 最终的输出是一个长度为 N 的向量 $y=\{y_1, y_2, y_3, \ldots, y_N\}$, 其中, N 代表可能的字符总数. 向量的每个标签 y_i ($i\in[1, N]$) 表示了对应字符的概率预测, 取值范围在 [0, 1] 之间. 训练这样的编码器是一个单分类任务, 标签采用被预测字符的 one-hot 编码. 我们采用交叉熵损失函数, 并使用默认设置的 Adam 优化器来训练模型.

3.2.2 模型训练

训练阶段的主要目的是提高模型对于压缩数据的预测能力. 基于神经网络的压缩方法通常可以分为自适应、静态和半自适应这 3 种. 自适应的神经网络压缩方法通常使用相同的随机模型进行初始化, 在压缩和解压的过程基于局部数据进行自适应更新. 但这种方法由于缺失全局视野, 当输入的数据发生变化时, 预测精度会大幅下降. 此外, 由于模型的初始参数是随机的, 它对于序列中靠前部分的预测表现通常较差. 静态的神经网络压缩在压缩前对所有数据进行训练, 并在压缩和解压阶段使用训练完成的静态模型. 然而, 这种方法在每次压缩时都需要重新训练模型, 在多数场景中并不适用. 我们注意到, 微服务系统的追踪数据结构在一定时期内通常不会发生很大的变化, 而且在经过预处理后跨度数据因为去除了追踪 ID、时间戳、父跨度 ID 等动态数据, 其内容高度冗余. 这使得我们可以使用半自适应的方法, 即使用少量数据进行训练并保存模型的参数, 在压缩时加载该微服务系统对应的模型, 并在压缩时对模型参数进行动态更新. 这种方式可以提高方法针对特定系统追踪数据的预测精度, 同时不需要很高的时间成本.

为了训练编码器, NCQT 首先对每个分组的数据按照一定数量进行随机采样, 并将其拆分为大小为 32 的窗口序列. 然后, 以 2 048 的批次大小进行分批训练, 学习率为 0.005, 共计训练 3 个轮次. 训练完成后, 该模型将作为对应微服务系统的初始模型, 在后续压缩和解压中使用. 需要注意的是, 即使训练的编码器无法对压缩的数据实现较高的预测精度, 它依旧可以用于数据的无损压缩和解压. 拥有较好预测结果的模型通常会带来更好的压缩效果, 反之亦然.

3.2.3 压 缩

在获取训练完成的编码器后, NCQT 将其与算术编码结合来对字符进行编码. 完整的压缩过程如算法 2 所示. 首先, 将数据拆为长度为 c 的窗口 (默认为 32). 由于第 1 个窗口的数据无法使用前 c 个字符预测其概率, 所以将其按照固定的均匀分布概率由算术编码生成表示. 然后, 使用训练后的模型对每个窗口的下一个字符进行预测, 并得到相应的概率估计. 生成的概率和下一个字符将输入到算术编码器中更新数据的编码区间. 以上步骤循环进行直至所有字符完成预测与编码.

算法 2. 压缩过程.

输入: 输入序列 $x = \{x_0, x_1, x_2, \ldots, x_{end}\}$, 窗口长度 c;
输出: 压缩文件.

1. $i \leftarrow 0$
2. **while** $0 \leqslant i < c$ **do**
3. *Arithmetic Encode*(x_i, 1/23) // 以固定概率编码前 c 个字符
4. $i \leftarrow i + 1$
5. **end**
6. *Model.initialize*() // 初始化模型
7. **while** $c \leqslant i \leqslant end$ **do**
8. $P(x_i|x_{i-c}, \ldots, x_{i-1}) \leftarrow$ *Model.forward*($x_{i-c}, \ldots, x_{i-1}$) // 计算条件概率
9. *Arithmetic Encode*(x_i, $P(x_i|x_{i-c}, \ldots, x_{i-1})$) // 编码第 i 个字符
10. *Model.backward*(x_i, $P(x_i|x_{i-c}, \ldots, x_{i-1})$, *loss function*) // 根据预测结果动态更新模型参数
11. $i \leftarrow i + 1$
12. **end**

窗口长度 c 决定了模型预测时的上下文长度. 本文提出的方法采用神经网络模型进行数据压缩, 这时如果设置的窗口太小, 则可能会导致模型无法充分利用上下文信息, 难以捕捉到长距离的依赖关系, 导致预测效果降低. 同时, 窗口太小会使得方法需要更多的循环次数来处理整个序列, 可能会导致整体效率降低. 如果设置的窗口太大, 较大的输入维度会使模型训练更加复杂, 可能需要更长的训练时间和更多的计算资源, 而且容易过拟合. Bellard 等人[31]对不同的窗口大小进行了对比, 较小的窗口长度下的模型参数量更少, 预测时间较短且资源消耗较低, 而较大的窗口大小下模型通常具有更高的预测准确率. 在实际使用过程中, 应该综合考虑资源数量以及待压缩追踪数据的平均长度选择合适的窗口大小, 以权衡资源消耗、模型泛化能力以及模型的上下文捕捉能力.

在预测过程中, 神经网络生成每个字符在前置窗口下的条件概率. 在编码过程中, 算术编码将待编码的数据映射到一个位于 (0, 1) 的实数区间中, 并输出一个位于区间内的小数以表示全部数据. 具体而言, 编码器初始设置实数区间 (L, H) 为 (0, 1) 以存储中间结果, 其中 L 为区间的下界, H 为区间的上界. 对于第 i 个需要编码的字符 s_i, 根据预测过程生成的概率统计表 $y^i = \{ y_1^i, y_2^i, \ldots, y_N^i \}$进行编码, 其中 N 为字符的数量. 然后, 通过应用以下公式进行编码:

$$\begin{cases} \arg\left(y_t^i\right) = s_i,\ o_j^i = \dfrac{\sum_{k=1}^{j} y_k^i}{\sum_{k=1}^{N} y_k^i},\ i \in \{1, \ldots, n\} \\ L_i = L_{i-1} + (H_{i-1} - L_{i-1}) \cdot o_{t-1}^i \\ H_i = L_{i-1} + (H_{i-1} - L_{i-1}) \cdot o_t^i \end{cases} \tag{2}$$

其中, n 为总字符数, t 表示 s_i 在统计表中的位置. 在对所有 n 个字符执行上述过程后, 编码器可以得到一个最终的

区间 (L_n, H_n). 最后, 算术编码在该区间内选取一个具有最短的二进制表示的数字, 作为所有输入的最终表示.

为了更好地预测被压缩分组的数据, NCQT 在压缩过程动态更新模型. 为了控制额外的时间开销, 我们设置了更新时间步 T. 模型每迭代 T 次会将前 T 个窗口拼接并计算整体预测的损失, 然后通过反向传播更新模型的参数, 并在下个时间步内使用该模型进行预测. 这个优化实际上是压缩比和压缩时间之间的平衡, 我们将在第 4 节研究这种优化对压缩的影响.

除此之外, 模型预测和算术编码被设计为并行运行以提高方法压缩效率. 我们注意到, 算术编码器需要模型提供概率, 然而模型的下一轮预测无需算术编码的结果. 因此, NCQT 在压缩时启动了模型预测和算术编码两个子进程. 模型在完成预测后将结果加入任务队列中, 而算术编码进程则不断从队列中取出结果并生成下一区间的数字表示. 当所有字符预测完成时, 模型预测进程向任务队列中加入结束符号, 算术编码进程在接收到结束符号后输出最终的结果.

3.3 追踪数据搜索和解压

在冗余抽取阶段中, NCQT 在预处理时记录了追踪结构字典以及每种跨度的字段索引字典和组合字典, 并在分组时记录了分组数据的索引字典. 这些索引表所占空间较小, 且对提高查询速度有显著作用. 因此, 索引表并不会随跨度数据一同交由神经网络压缩, 而是使用轻量压缩工具 gzip 简单打包. 在搜索阶段, NCQT 首先将压缩的索引表解压, 然后判断压缩数据中是否存在满足用户查询条件的跨度数据, 并定位到数据所在的分组, 最后解压涉及的分组并还原其中符合要求的追踪数据.

为了方便用户搜索, NCQT 参考 Grafana Tempo 的 TraceQL 查询语句[21], 设计和实现了用于查找追踪数据的查询语句, 支持用户对压缩数据执行各种复杂查询. 目前, NCQT 支持根据追踪 ID、跨度或资源的字段、时间和结构等搜索满足条件的追踪, 表 1 给出了不同查询条件下查询语句的语法规则. 与现有工具类似, 追踪 ID 查询允许用户根据全局唯一的追踪 ID 查询追踪. 在字段查询中, 用户可以根据资源或跨度的属性字段筛选追踪. NCQT 支持通过比较运算符 (包括>、<、>=、<=、!=等) 限定字段的取值范围, 并允许用户使用多组查询条件. 为了方便用户了解追踪数据的取值特征, NCQT 提供了压缩数据上的聚合查询. 用户可以选择资源或跨度的某个字段, 查询该字段在压缩数据中取值的范围、最大值和最小值等信息. 考虑到追踪数据包含了系统请求的调用过程, NCQT 还提供了结构查询的功能. 用户可以指定追踪的跨度关系, 并可以查询某一跨度在调用链上下游中几跳内的相邻跨度.

表 1 查询语句的语法规则

查询条件	查询语句正则表达式
追踪标识符查询	Query trace by traceID: '([^']*)'
属性字段查询	Query (span|trace) by (span|resource) fields: (.+)
字段取值范围查询	Query value range for (span|resource) field: '([^']*)'
字段最值查询	Query (max|min) value for (span|resource) field: '([^']*)'
上下游关系查询	Query trace based on context relation: '([^']*)' (follows|precedes) '([^']*)'
上下游节点查询	Query (upstream|downstream) nodes for span: '([^']*)' hops (\d+)

3.3.1 搜 索

给定一个字段查询语句, NCQT 将其解析并转换为需要满足的条件组, 然后交由定位器找到满足条件的分组. 算法 3 和图 8 给出了一个单字段搜索的示例, 定位器首先查询每个跨度容器的索引字典, 以确认目标字段是否存在于该跨度中. 如果存在, 定位器记录字段对应的参数字典中满足条件的参数值的索引. 其次, 定位器在组合字典中寻找包含目标字段的组合, 并记录目标字段在该组合中的位置. 接着, 定位器在结构容器中找出所有包含目标跨度的结构. 由于预处理后的追踪数据按照结构的前序遍历顺序依次存储跨度, 因此可以根据结构额外记录目标跨度在所在结构的追踪中的存储位置. 然后, 定位器在每个分组的索引中检查该分组是否包含目标结构, 将满足条件且尚未解压的分组加入解压列表. 最后, 定位器在解压后的分组中找到目标追踪条目, 并根据之前记录的信息快速

判断目标跨度的目标位置的参数索引是否在符合要求的取值索引中. 如果符合条件, 则交由重构器还原该追踪的文本表示.

算法 3. 单字段搜索过程.

输入: 目标字段 *key*, 比较符 *operator*, 比较取值 *value*, 跨度容器 *span_containers*, 结构列表 *structs*, 压缩分组 *groups*;
输出: 符合条件的追踪数据 *traces*.

```
1. spans ← {}, structs ← {}, traces ← []
2. for each span in span_containers //遍历所有跨度容器, 寻找符合条件的跨度
3.     if key in span.key_dict and index_of(value, span.param_dict[key], operator, value) > 0
4.         spans[span] ← index_of(value, span.param_dict[key], operator, value) //记录符合条件的取值索引
5. for each struct in structs //遍历所有结构, 寻找包含目标跨度的结构
6.     if struct.spans ∩ spans is not empty
7.         structs[struct] ← position_of(struct, spans) // 记录目标跨度在结构中的位置
8. for each group in groups
9.     if group.structs ∩ structs is not empty
10.        data ← decompress(group) // 解压符合条件的压缩分组
11.        for each trace in data
12.            if trace.struct ∈ structs and trace.get_value(structs[trace.struct]) ∈ spans
13.                traces ← trace ∪ traces //将符合条件的数据加入结果列表中
14. return traces
```

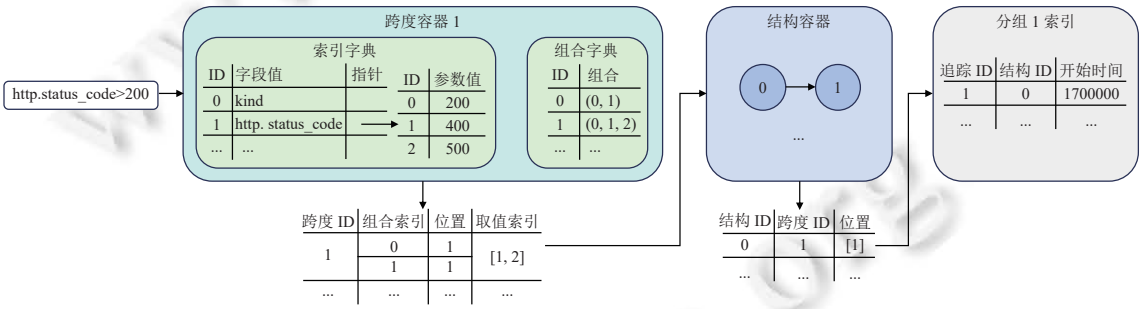

图 8 字段搜索示例流程

除了字段查询以外, NCQT 还支持根据结构搜索相关追踪. 由于结构容器中记录了每种结构的序列化表示, 定位器既可以快速判断被查询的两个跨度是否在某个结构中属于祖先与后代的关系, 又可以找出被查询跨度指定跳数以内的上下游节点. 相较于字段搜索, 结构搜索无需遍历跨度容器, 因此拥有更快的查询速度.

NCQT 通过预加载的方式缓解了搜索冷数据的问题. 如果查询过程中定位到的分组尚未解压, 则会带来较长的解压时间开销, 第 4 节给出了分组是否需要解压时的查询延迟对比. 考虑到用户在查询数据时可能围绕同一场景对相关追踪进行搜索, 且压缩的追踪数据根据相似性分组, NCQT 在解压时可以选择对被解压分组的相邻分组后台并行解压, 从而提高后续查询命中已解压分组的概率.

3.3.2 解 压

从编码器解压数据的过程与压缩过程相似. 算术编码使用网络模型的概率估计解析数据, 然后网络模型根据新窗口输出下个字符的概率, 重复该过程直至所有数据解压完成. 具体而言, NCQT 首先使用相同的均匀分布概率

解码前 c 个字符, 然后对后续所有字符, 遵循以下流程依次解码.

(1) 模型利用 c 个字符的窗口预测下一个字符的概率分布.

(2) 算术编码利用概率估计解码下一个字符.

(3) 向后滑动窗口, 用于下一轮的模型预测.

在解码过程中, 算术编码按照编码时相同的概率统计表划分 (0, 1) 的实数区间, 并根据编码数字所落的子区间输出对应的符号. 然后, 选择对应的子区间, 并在选择的子区间上继续执行下一轮的划分和解码过程. 具体公式如下:

$$\begin{cases} o_j^i = \dfrac{\sum_{k=1}^{j} y_k^i}{\sum_{k=1}^{N} y_k^i},\ s_i = \arg\left(y_r^i\right),\ i \in \{1,\ldots,n\} \\ L_i = L_{i-1} + (H_{i-1} - L_{i-1}) \cdot o_{r-1}^i \\ H_i = L_{i-1} + (H_{i-1} - L_{i-1}) \cdot o_r^i \end{cases} \tag{3}$$

其中, r 表示编码数字在第 i 次解码时所处子区间的位置. 解码的过程相当于编码过程的逆运算, 算术编码重复执行上述过程逐步确定编码前的符号, 直至所有符号完成解码.

为保证无损解压, 解压时使用压缩时相同的初始网络模型和同样的随机数种子, 并在 T 个时间步后通过反向传播更新模型参数.

编码器解压后的数据是追踪数据的数字表示, 需要进一步还原为原始的文本表示. 重构器在这一步执行预处理阶段的反向操作, 方法首先利用组合字典和索引字典还原字段的键值表示, 然后根据追踪的开始时间为每个跨度重新计算开始与结束时间, 最后从对应的结构中还原出跨度之间的调用关系.

4 实验分析

为了验证本文所提出方法的有效性, 我们使用 Python 实现了 NCQT, 并使用来自多个开源微服务系统的追踪数据进行评估. 本节主要围绕以下 3 个问题展开实验验证.

- RQ1: 本文提出的方法与通用压缩算法相比压缩效果和效率如何?
- RQ2: 本文提出的方法与追踪查询工具相比存储开销和查询性能如何?
- RQ3: 相关超参数和可选优化对实验结果有何影响?

4.1 实验设置

本节所有实验均在 Ubuntu 20.04 机器上进行, 该机器配有 36 核 CPU、125 GB 主存以及 4 块 24 GB 的 NVIDIA GeForce RTX 3090 显卡.

4.1.1 实验数据

根据以往研究的经验, 我们选择了 Train Ticket (https://github.com/FudanSELab/train-ticket)[49,50]、Sock Shop (https://github.com/microservices-demo/microservices-demo) 、Robot Shop (https://github.com/instana/robot-shop) 和 Astronomy Shop (https://github.com/open-telemetry/opentelemetry-demo) 这 4 个不同的开源微服务系统用于生成评估数据集. 在 GitHub 的开源微服务系统中, 以上系统使用了多种开发语言, 拥有较多的微服务数量, 且服务间的调用关系相对复杂.

实验使用 OpenTelemetry 生成分布式追踪数据. 具体而言, 我们使用 OpenTelemetry Operator 为每个服务注入和配置 OpenTelemetry 自动插桩库, 并将微服务系统部署在一个含有 4 个节点的 Kubernetes 集群上, 每个节点拥有一个 8 核 CPU 和 15 GB 内存. 为了收集和存储追踪数据, 我们在集群中部署了 OpenTelemetry Collector 和 Grafana Tempo.

表 2 展示了从 4 个微服务系统中收集到的数据的详细信息, 这些数据集中的追踪数据是通过负载生成器模拟不同用户请求生成的. 已有的研究发现, 神经网络压缩大文件往往时延较长, 于是我们构建了较小的数据集进行初

步实验. 在小数据集的生成上, 我们使用同样的方式部署微服务系统和发送请求, 唯一的不同是我们缩小了收集追踪数据的时间窗口以减少数据量.

表 2 实验数据集

微服务系统	语言	服务数量	追踪数量	大小
Train Ticket	Java, NodeJS, Python, Go	41	134 143 13 000	9.8 GB 975 MB
Sock Shop	Java, NodeJS, Go	14	187 701 31 881	4.1 GB 702 MB
Robot Shop	Java, NodeJS, Python, Go, PHP	12	519 975 72 000	6.0 GB 851 MB
Astronomy Shop	Java, NodeJS, Python, Go, PHP.NET, C++, Kotlin, Ruby, Rust	14	491 125 80 568	5.8 GB 981 MB

4.1.2 评估指标

为了衡量压缩的效果, 我们使用压缩比 (*CR*) 作为评估有效性的指标. 它被广泛用于压缩方法的评估. 其定义如下:

$$CR = \frac{\text{原始文件大小}}{\text{压缩后文件大小}} \tag{4}$$

在给定数据集中, 原始文件大小是固定的, 不同方法生成的压缩后文件大小不同. 压缩后的文件越小, 压缩比越大, 表示压缩更有效.

4.1.3 对比方法

在压缩方面, 我们使用传统压缩算法和基于神经网络的压缩算法与本文提出的 NCQT 方法对比. 具体而言, 我们选择了 gzip、bzip2 和 lzma 这 3 种传统压缩算法, 同时引入了两种最新的神经网络压缩方法进行对比.

● TRACE[33]. TRACE 是一种基于单层 Transformer 的无损压缩算法, 它引入 Transformer 以并行地构建历史依赖关系, 并设计了一个控制器降低参数更新开销以加快压缩过程.

● DZip[27]. DZip 是一种基于 GRU 的序列数据压缩算法, 它采用了一种基于自适应和半自适应训练的混合架构, 在多种数据类型上压缩效果出色.

在搜索方面, 我们将 NCQT 与现有的追踪查询工具进行比较, 具体工具如下.

● Jaeger[22]. Jaeger 是一个开源的分布式追踪系统, 其中的 Jaeger Query 提供了接口, 支持用户根据时间范围、服务、字段等查询分布式跟踪数据.

● Grafana Tempo[21]. Grafana Tempo (简称 Tempo) 是一个开源的分布式追踪后端, 专注于长期存储和检索大规模分布式追踪数据. Tempo 实现了追踪查询语句 TraceQL, 允许用户精确、高效地查询满足指定条件的追踪或跨度.

4.2 与通用压缩算法的压缩效果和效率对比 (RQ1)

为了研究 NCQT 的压缩效果, 我们使用 NCQT 和对比方法对 4 个微服务系统收集的追踪数据集进行了压缩. 我们采用 Transformer 和 GRU 两种网络结构作为编码器, 即 NCQT-Transformer (简写为 NCQT-T) 和 NCQT-GRU (简写为 NCQT-G). 为了确保对比实验的公平性, NCQT-T 采用了与 TRACE 相同的网络结构, 而 NCQT-G 采用了与 DZip 相同的网络结构, 模型参数均为默认值.

我们首先在小文件数据集上进行了初步验证, 结果如表 3 所示. 我们统计了所有方法压缩后的文件大小, 其中对比方法仅产生单个压缩文件, 而 NCQT 的压缩文件由索引表和压缩的追踪数据两部分组成, 除此之外, 我们还记录了冗余抽取阶段后所有中间文件的大小. 在传统压缩算法中, gzip、bzip2 和 lzma 平均可以达到 19.8、31.5 和 40.0 的压缩比, 其中, lzma 在所有数据集上均展现最佳的压缩效果. TRACE 和 DZip 在 4 个数据集中分别可以达到平均 41.4 和 42.1 的压缩比, 在多数数据集上, DZip 比 TRACE 压缩比更高, 但在 Robot Shop 数据集上, TRACE 表现出更好的压缩效果. 通过对比发现, 基于神经网络的压缩方法在多数数据集上优于传统压缩算法. 以 Train

Ticket 数据集为例, TRACE 和 DZip 的压缩比是 gzip 的 1.98 倍和 2.02 倍, 对于效果最好的传统压缩算法 lzma, TRACE 和 DZip 也有 4%–6% 的效果提升. 这证明了基于神经网络的压缩算法相较于传统压缩算法有更高的压缩上限, 这是因为传统压缩算法虽然可以识别数据中的局部冗余, 但其缺乏对数据集整体数据分布的理解, 而神经网络压缩算法可以更好地学习数据集中的整体信息, 从而实现更高效的压缩.

表 3　不同方法对小文件数据集的压缩结果

方法	Train Ticket		Sock Shop		Robot Shop		Astronomy Shop	
	大小 (MB)	压缩比	大小 (MB)	压缩比	大小 (MB)	压缩比	大小 (MB)	压缩比
原始	975	1	702	1	851	1	981	1
gzip	47.4	20.6	35.0	20.1	38.9	21.9	58.7	16.7
bzip2	32.7	29.8	19.3	36.4	28.0	30.4	33.2	29.5
lzma	24.9	39.2	15.1	46.5	25.0	34.0	24.4	40.2
TRACE	23.9	40.8	14.5	48.4	23.7	35.9	24.3	40.4
DZip	23.4	41.7	14.1	49.8	26.0	32.7	22.2	44.2
冗余抽取	71.8	13.6	28.3	24.8	74.0	11.5	58.3	16.8
NCQT-T	5.4	180.6	**3.8**	**184.7**	4.6	185.0	9.3	105.5
NCQT-G	**5.2**	**187.5**	**3.8**	**184.7**	**4.5**	**189.1**	**9.2**	**106.6**

在所有数据集上, NCQT 均获得了最高的压缩比. 从实验结果可以看出, NCQT-T 和 NCQT-G 的压缩比远高于传统压缩算法和基于神经网络的对比算法. 在 Robot Shop 数据集上, NCQT-T 的压缩比是 TRACE 的 5.15 倍, 而 NCQT-G 的压缩比是 DZip 的 5.78 倍, 这意味着 NCQT 相较于两个对比方法可以将数据缩小至 1/5 甚至更低. 我们注意到, NCQT 在 Astronomy Shop 数据集上效果较差, 仅有 105.5 和 10.6.6 的压缩比. 这是由于 Astronomy Shop 是 OpenTelemetry 开源的示例微服务系统, 服务中定义了丰富的字段种类, 导致 NCQT 产生了较大的索引字典, 降低了整体的压缩比. 此外, 通过对比冗余抽取方法和对比方法的实验结果可以发现, NCQT 在冗余抽取阶段已经接近甚至超过了部分传统压缩算法的压缩效果, 这是因为冗余抽取方法通过预处理和聚类分组消除了追踪数据中存在的大量冗余信息. 因此, NCQT 在冗余抽取和神经网络压缩的双重作用下获得了最高的压缩比. 实验结果表明, NCQT 与现有的通用压缩算法相比, 在追踪数据压缩方面有明显的优势, 并且能很好地泛化到不同微服务系统产生的追踪文件中, 这是因为对比方法并未针对追踪数据的特点进行优化, 而 NCQT 在设计上充分考虑了追踪数据中存在的模式冗余和结构冗余.

我们进一步在大文件数据集上进行了评估, 结果如图 9 所示. 值得注意的是, TRACE 和 DZip 压缩大文件的耗时会超过 80 h, 不符合实际需求, 因此本文并没有将其纳入大文件压缩的对比方法中. 在大文件数据集上的观察结果与小文件数据集基本一致. 对于每个微服务系统产生的追踪数据, 不同的压缩方法拥有与表 3 相似的压缩比. 我们注意到, NCQT 在大文件数据集上的压缩比甚至更高, 这是由于随着追踪数量的增加, 索引字典的大小也会逐渐增长, 但是增长速度相对缓慢, 压缩后的字典在压缩文件中的空间占比逐渐减少. 实验结果表明, NCQT 在大数据集上也能高效地压缩追踪数据.

最后, 我们对不同压缩方法的效率进行了评估. 传统压缩算法的时间开销只有压缩和解压两部分组成; 对于基于神经网络的通用压缩算法, 我们测量了他们的模型训练时间、压缩时间和解压时间; 而 NCQT 的运行时间可分为冗余抽取、模型训练、压缩和解压这 4 个部分. 由于 TRACE 和 DZip 压缩效率较低, 我们只显示小文件数据集上的结果. 实际上, 在相同配置下, 压缩方法的时间开销基本上与压缩文件大小呈线性关系, 因此在小数据集上也可以测量不同方法的相对效率. 我们在表 4 展示了不同压缩算法的时间成本, 并在图 10 展示了基于神经网络的压缩算法在不同阶段的时间对比.

从表 4 可以看出, 传统的压缩算法在时间上具有显著的优势. 这是由于传统压缩算法采用滑动窗口识别重复出现的序列, 具有线性的时间复杂度, 而神经网络压缩算法中每个窗口都需要经历模型的前向和反向传播, 引入了大量的时间开销. 其中 gzip 表现最为出众, 在最大的 Astronomy Shop 数据集上仅需要 25 s 的压缩时间和 2 s 的解压时间. 因此, NCQT 选择 gzip 作为索引字典的压缩算法, 以减少查询时需要等待的解压时间.

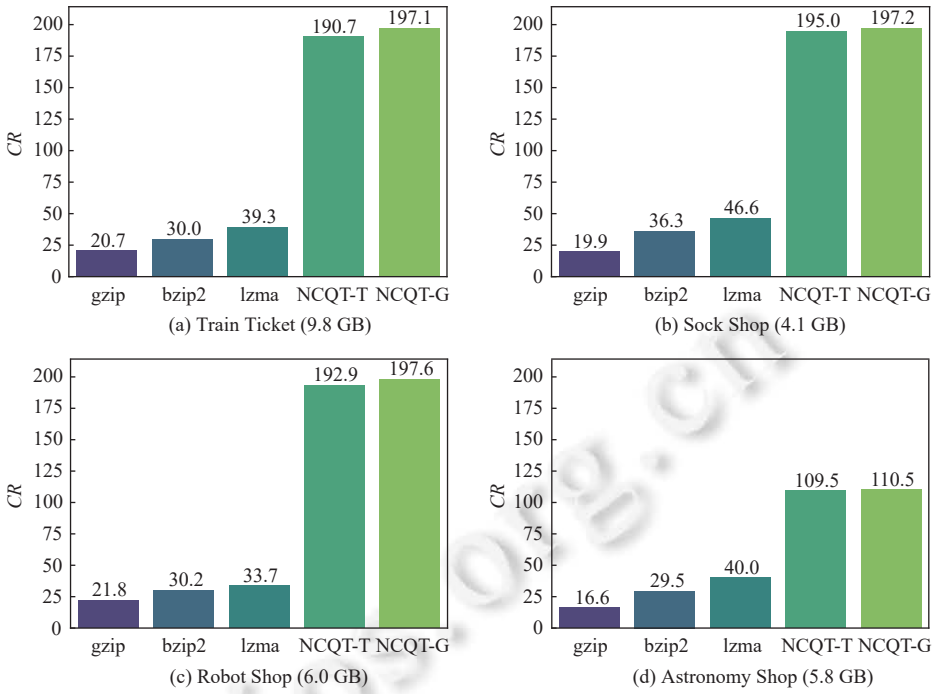

图 9 不同大文件数据集压缩效果对比

表 4 不同压缩算法的时间成本 (s)

方法	Train Ticket		Sock Shop		Robot Shop		Astronomy Shop	
	压缩	解压	压缩	解压	压缩	解压	压缩	解压
gzip	16	3	11	2	19	2	25	2
bzip2	137	17	107	17	130	18	141	22
lzma	91	3	61	3	88	3	101	3
TRACE	60 916	57 939	46 491	32 673	52 656	37 901	62 238	41 598
DZip	56 096	27 942	39 789	21 612	53 309	26 776	58 888	27 309
NCQT-T	1 003	1 036	440	434	573	435	447	379
NCQT-G	803	738	344	316	430	290	347	291

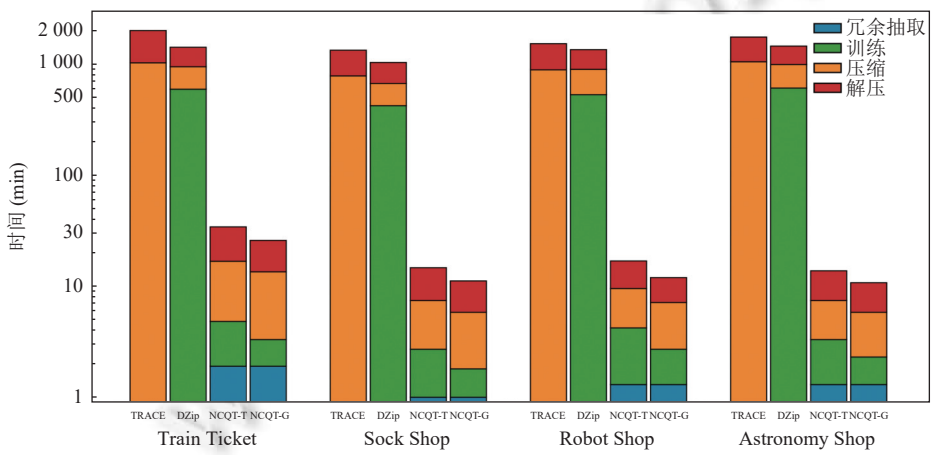

图 10 基于神经网络的压缩算法效率对比

图 10 则展示了基于神经网络压缩算法的时间对比, 每组柱状图从左到右分别是 TRACE、DZip、NCQT-T 和 NCQT-G. 需要注意的是, 纵坐标以 min 为单位, 并且采用非均匀刻度, 这是由于对比方法的时间开销均在 1 000 min 以上, 而 NCQT 的运行时间基本都在 30 min 以内. 实验发现, NCQT 在 4 个数据集上的效率是 TRACE 和 DZip 的 41.2–161.7 倍, 这表明 NCQT 在压缩追踪数据上比最先进的神经网络压缩算法更加高效, 这是我们设计的优势. 通过利用追踪数据的特征, NCQT 在提取冗余后将文本数据转换为数字格式, 不仅减少了模型的输入, 而且减少了数据的词汇量, 因而能更高效地压缩和解压. 同时, NCQT 将模型预测与算术编码拆解为两个并行的子进程, 大幅度减少了压缩过程中的 I/O 等待时间

通过对比每个步骤的时间, 我们发现 NCQT 在冗余抽取步骤花费的时间平均不足 2 min, 与其他步骤相比, 耗时几乎可以忽略不计, 但是可以带来显著的压缩效果提升. 除此之外, 基于神经网络的压缩算法均花费大量时间解压数据, 几乎占整个时间开销的一半. 根据已有工作[51]的观察和分析, 在基于算术编码的方法中, 解码阶段通常需要较长的时间, 因为需要搜索数字所在区间以获取正确编码. 另一方面, 压缩阶段的模型预测无需算术编码的结果, 但解压阶段模型需要算术编码解码出下一个字符以组成新的预测窗口, 算术编码也需要模型产生的概率估计解码字符, 这种循环依赖关系导致了我们无法通过并行算法加速解压过程.

通过对比不同数据集上的时间开销, 我们发现 NCQT 在 Train Ticket 数据集上用时更长, 尽管该数据集大小并未与其他数据集存在明显差异. 主要原因是, Train Ticket 服务数量较多, 部分请求的调用链相对较长. NCQT 在分组时将追踪数据按照数量划分为 4 组, 并且倾向于将相似的长追踪分为一组, 从而导致其中一个分组的数据长度远高于其他分组, 无法充分利用并行压缩提高压缩和解压效率. 在第 4.4 节中, 我们将进一步研究分组参数对压缩效率的影响.

4.3 与追踪查询工具的存储开销和查询性能对比 (RQ2)

在本节中, 我们将 NCQT 与常用的追踪存储和查询工具 Tempo 以及 Jaeger 进行了比较. 为了比较的公平性, 我们将 Tempo 和 Jaeger 以单机的形式部署在服务器上. 其中, Tempo 采用本地存储的方式保存追踪数据, 而 Jaeger 则使用内存存储组件保存数据, 我们通过工具的 OTLP receiver 将数据导入其中. 在本节的实验中, 我们默认选择 GRU 作为 NCQT 编码器的网络模型.

我们首先对比了不同工具在各数据集上的存储成本, 其结果如表 5 所示. 结果显示, Tempo 和 Jaeger 在所有数据集上的存储开销均高于 NCQT. 其中, Jaeger 存储追踪数据所占空间最大, 平均是 Tempo 的 31 倍, 是 NCQT 的 673 倍. 尽管 Tempo 的空间占用小于原数据文件大小, 但仍明显高于 NCQT. 对比图 9 的实验结果, 我们发现 Tempo 和 Jaeger 与通用压缩算法相比需要更多的存储空间, 这是因为追踪查询工具为了便于查询为追踪数据建立了索引, 传入的追踪文件越大, 建立索引所需的空间也会越多. 实验结果证明了这些追踪查询工具以牺牲存储为代价实现对追踪数据的查询支持.

表 5 不同工具的存储成本比较 (MB)

工具	Train Ticket	Sock Shop	Robot Shop	Astronomy Shop
原始	9 945	4 134	6 075	5 895
Tempo	1 156.4	411.1	929.4	858.6
Jaeger	44 707.1	15 134.1	24 009.9	20 991.2
NCQT	50.5	21.0	30.8	53.4

为了评估 NCQT 在查询上的性能, 我们使用表 6 中的一组查询对每个工具进行了测试, 这些查询特定于在 Robot Shop 系统中收集到的 6.0 GB 数据集. 在设计查询集时, 我们综合考虑了 Tempo 和 Jaeger 支持的查询接口, 并参考了 Tempo 提供的追踪查询语句 TraceQL. 查询集最终定义了 4 种查询类型, 包括追踪 ID 查询、字段查询、聚合查询以及结构查询. 除了追踪 ID 查询以外, 我们对于每种查询类型设计了多条不同的查询语句, 表中后 3 列对应查询结果涉及的追踪数、跨度数和结构种类. 对于表 6 中的每条查询, 我们将其转换为每种工具对应支持的查询语句, 并取 10 次运行的平均值作为结果. 由于 Jaeger 未提供聚合查询或结构查询的接口, 因此我们将其在对应查询的结果置空.

表 6 用于查询性能评估的查询集

查询类型	序号	查询语句	追踪	跨度	结构种类
追踪ID查询	Q1	Query trace by traceID: 'twMveyd0doAiUYlrTcG7Fg=='	1	9	1
字段查询	Q2	Query span by resource fields: 'service.name'='shipping' and 'duration'>'500ms'	6 207	21 553	21
	Q3	Query trace by resource fields: 'k8s.node.name'='k8s-node-3' and 'telemetry.sdk.language'!='nodejs'	14 190	88 528	21
	Q4	Query span by span fields: 'db.system'='mysql' and 'thread.name'='http-nio-8 080-exec-36'	52	52	12
	Q5	Query trace by span fields: 'name'='GET /check/:id' and 'http.status_code'>='400' and 'startTimeUnixNano'>'1 701 790 447 000 000 000'	359	7 180	4
聚合查询	Q6	Query max value for span field: 'http.response_content_length'	—	—	—
	Q7	Query value range for resource field: 'process.command'	—	—	—
结构查询	Q8	Query trace based on context relation: 'shipping#SELECT City' follows 'shipping#CityRepository.findByCode'	2 838	14 166	7
	Q9	Query trace based on context relation: 'shipping#HTTP POST' precedes 'cart#middleware-query'	2 445	34 228	2
	Q10	Query upstream nodes for span: 'catalogue#dns.lookup' hops 3	5	14	5
	Q11	Query downstream nodes for span: 'user#GET' hops 4	542	1 086	2

表 7 显示了 NCQT 和其他工具的搜索性能对比. 由于 NCQT 在查询的过程中可能涉及解压, 因此我们分别列出了所有数据无需神经网络解压 (NCQT-Warm) 以及所有数据均需神经网络解压 (NCQT-Cold) 时的时间开销. 从表 7 中可以看出, 在不需要解压的情况下, NCQT 在所有类型中的查询开销均为最低值. 在 Q2 的查询中, NCQT-Warm 的查询效率是 Tempo 的 3 776 倍, 是 Jaeger 的 40 倍; 而对于 Q1 单条追踪的查询, NCQT-Warm 与对比工具也拉开了 10 倍和 3 倍的差距; 在结构查询上, Tempo 的时间开销平均是 NCQT-Warm 的 128 倍. 结果表明, NCQT 在不需要解压时查询速度快于 Tempo 和 Jaeger, 这是由于 NCQT 在冗余提取时提供了丰富的索引字典, 搜索时无需对所有数据进行扫描, 而且可以快速判断追踪数据是否满足要求. 然而, 当满足条件的数据未被解压时, NCQT 的查询性能会有所降低, NCQT-Cold 显示了所有数据均需要解压的最坏情况下的查询时间. 相较于现有的追踪查询工具, NCQT 在该情况下平均需要花费更多的时间查询数据. 然而, NCQT-Cold 在 Q3 上优于 Tempo, 这意味着当满足条件的追踪数据较多时, NCQT 可以持平甚至超过 Tempo 的查询速度. 实验结果证明了冷数据问题对 NCQT 查询效率的影响, 因此 NCQT 在搜索时会对目标分组的相邻分组预先解压, 以最大程度缓解解压带来的时间开销. 需要注意的是, 解压带来的额外成本仅在首次未命中时存在, 一旦分组解压后, 后续所有基于该分组查询的效率均与 NCQT-Warm 类似. 一般情况下, NCQT 的查询效率根据需要解压的分组数量介于 NCQT-Warm 与 NCQT-Cold 之间, 而分组指标会影响满足查询条件的数据在分组中的分布情况, 因此我们在第 4.4 节中进一步研究了分组对查询效率的影响.

表 7 不同工具的查询性能比较 (s)

查询	NCQT-Warm	NCQT-Cold	Tempo	Jaeger
Q1	0.007	226.730	0.071	0.020
Q2	0.095	601.410	358.747	3.860
Q3	4.278	604.368	820.161	7.928
Q4	0.041	599.332	2.201	0.759
Q5	0.334	599.752	18.939	1.379
Q6	0.003	0.003	0.090	—
Q7	0.001	0.001	1.605	—
Q8	0.771	598.407	141.047	—
Q9	1.311	488.842	122.458	—
Q10	0.021	758.967	1.400	—
Q11	0.161	344.563	27.520	—

此外, 我们发现 Jaeger 在 Q1–Q5 的查询效率均高于 Tempo. 一方面, 实验中 Jaeger 使用内存数据库存储数据和索引, 相较于 Tempo 而言节省了系统 I/O 时间. 另一方面, Tempo 通过 TraceQL 查询后返回的结果集并非完整的数据, 需要根据结果集中每条追踪的 ID 进行二次查询, 而 Jaeger 可以直接返回满足查询条件的完整追踪列表. 然而, Jaeger 需要更大的存储空间, 且无法支持对于追踪数据的复杂查询.

4.4 敏感性分析 (RQ3)

NCQT 有一些可能会影响压缩或查询性能的可配置参数, 分别是分组的大小、批处理的大小、模型的学习率以及是否动态更新. 为了研究这些参数如何影响方法的性能, 我们对其进行了敏感性分析实验. 具体而言, 我们首先从数据集中选取了 Sock Shop (4.1 GB) 和 Robot Shop (6.0 GB) 作为本节的实验数据. Sock Shop 数据集相对较小, 而追踪数据的长度平均较长; Robot Shop 数据集则相对较大, 而追踪数据相对较短, 且在第 4.3 节中定义了 11 条查询语句. 选择这两个数据集作为实验对象可以充分研究不同情况下我们的方法受参数影响的敏感程度. 对于上文提到的每种参数, 我们使用不同的参数值进行实验. 通过比较不同的配置, 我们可以评估每个参数对实验结果的影响. 与第 4.3 节相同, 本节选取 NCQT-G 作为实验方法, 我们后续验证了上述参数的设置对 NCQT-T 具有相似的影响.

图 11 展示了 NCQT 在不同分组大小下的压缩性能表现. 其中, 分组大小指的是每个分组包含的追踪数量.考虑到两个数据中追踪数据的平均长度不同, 我们在不同的数据集上选择了不同的分组大小取值范围. 在追踪长度较长的 Sock Shop 数据集上, 我们倾向于将分组大小设置为较小的取值; 反之, 我们在追踪长度较短的 Robot Shop 数据集上设置了较小的分组大小. 通过这种方式, 我们确保了 NCQT 在两个数据集上生成的分组数量范围几乎相同. 从图中可以看出, NCQT 的压缩比随分组大小的增长几乎没有太大变化, 这意味着 NCQT 的压缩效果对分组大小并不敏感. 我们观察到, 在不同的分组设置下, 无论是冗余抽取阶段产生的索引字典还是编码器压缩阶段产生的压缩文件, 二者的大小均未产生较大的变化. 这证明了我们冗余抽取阶段产生的中间表示是易于学习的, 无论每个分组有多少数据量, 模型都有较高的预测精度. 相较于压缩比, 压缩时间受分组大小的影响较为明显. 结果表明, 在实验设置的分组大小取值范围内, 压缩时间随分组大小的增大整体上呈上升趋势. 这是因为 NCQT 的压缩过程是并行进行的, 当分组大小较小时会产生较多的分组数量, 可以充分利用服务器多核与多卡的优势. 我们发现, 压缩时间相对分组大小并非严格单调. 当分组数量多于显卡数量时, 会出现多个分组的压缩进程抢占同一 GPU 资源的情况, 此时增加分组数量可能导致原本耗时最长的 GPU 分担了更多的压缩任务, 从而增加了整体的压缩时间. 需要注意的是, 结果并不意味着分组大小越小越有利于提高压缩效率. 受限于服务器 CPU、内存、显存以及文件句柄数量等资源, 过多的分组数量会带来严重的资源竞争, 反而会大幅降低压缩效率.

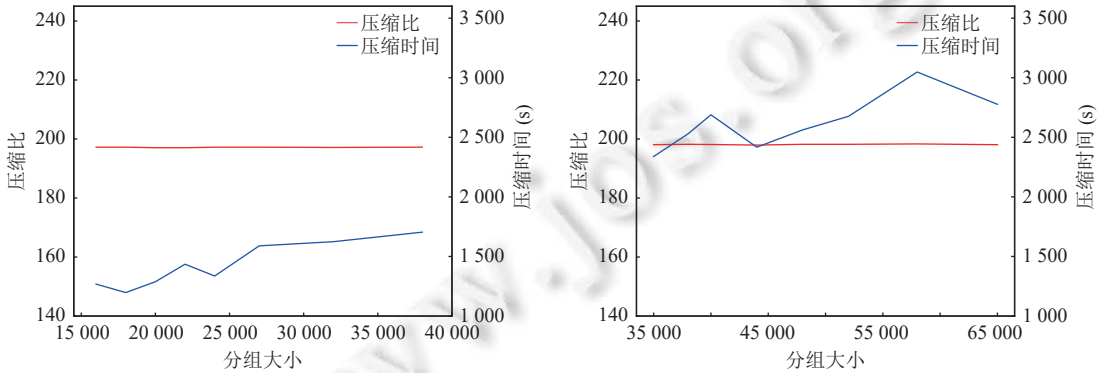

图 11 不同分组大小下的压缩性能变化

为了研究分组大小对查询性能的影响, 我们测试了不同分组大小下 NCQT 对 Robot Shop (6.0 GB) 数据集的查询效率, 结果如图 12 所示. 我们采用了第 4.3 节中定义的查询语句, 分别测量无需解压情况下 (NCQT-Warm) 以

及所有数据均需解压情况下 (NCQT-Cold) 的查询时间, 并取 11 条查询的平均值作为结果. 图 12(a) 展示了最优情况下的查询速度, 结果表明 NCQT 在查询命中受分组影响较小, 均能保证较高的查询效率. 从图 12(b) 中可以看出, 不同分组大小下 NCQT 解压目标分组的时间差异较大. 具体而言, 分组大小影响了被查询数据在分组中的分布, 因此解压时间实际上是由需要解压的分组数量与单个分组需要解压的数据量二者之间的平衡性决定的. 当分组较小时, 解压单个分组的速度较快, 但需要解压的分组数较多, 反之亦然. 例如, 分组大小在 35 000–40 000 时, 查询语句 Q2 和 Q3 涉及的分组数量为 5, 而分组大小 44 000–48 000 时, Q2 和 Q3 仅需解压 4 个分组, 因此前一段的平均解压速度慢于后一段. 同理, 虽然分组较大时仅需解压少量的分组, 但是目标分组中包含了大量的无关数据, 因而解压时间大幅提升.

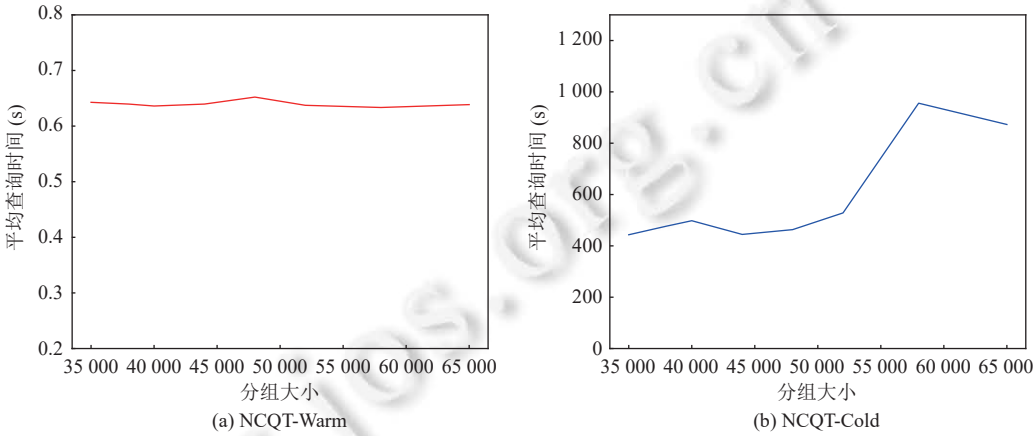

图 12 不同分组大小下的查询性能变化

批处理大小是一个典型的参数, 可以影响到模型预测的准确性与效率. 为了研究其影响, 我们在两个数据集上使用了不同的批处理大小, 并记录了对应的模型预测精度、压缩后文件大小以及压缩时间. 批处理大小为 512–4 096, 实验结果如图 13 所示. 总体而言, 模型的预测精度几乎不受分批大小变化的影响. 这主要因为我们冗余抽取阶段将自然语言文本转换为数值, 使得模型学习更加容易, 所以可以扩展到更大的批处理规模. 然而, 从图 13(b) 可以看出, 模型在相同预测准确率的情况下, 压缩效果却随批处理大小的增加而逐渐变差. 这是由于首个窗口无法借助模型预测, 批处理的每一个样本都需要以固定的概率编码第 1 个预测窗口的字符. 批处理规模越大, 无法利用模型进行预测的数据占比越高, 压缩效果越差. 另一方面, 图 13(c) 显示, 批处理大小影响模型的压缩时间. 以 Robot Shop 为例, 随着批处理大小从 512 增大到 4 096, 压缩时间从 1 877 s 减少到 1 200 s. 结果表明, 增加批处理大小可以显著提高压缩速度, 但当批量大小足够大后, 模型的压缩效率不会再增加, 这与之前的工作[52]结果一致.

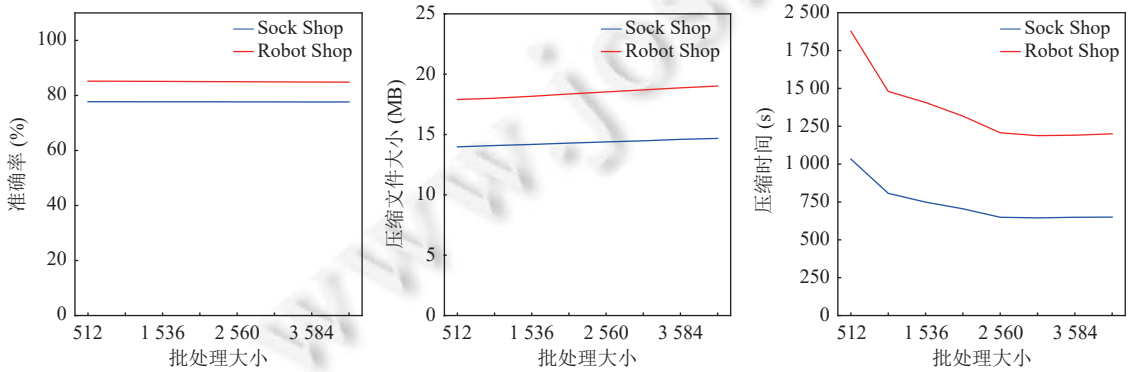

图 13 不同批处理大小下的压缩性能变化

学习率可以影响模型的预测精度, 从而影响最终的压缩文件大小. 为了测量其影响, 我们采用了从 0.1 到 0.000 01 共 5 种不同的学习率来训练和更新网络模型, 并记录 NCQT 在压缩过程的预测准确率以及压缩后的文件大小, 结果如表 8 所示. 从表中可以看出, 学习率较高时模型的预测准确率较低, 导致最终压缩文件的大小较大. 这是因为高学习率会导致优化过程容易跳过最优值, 从而导致模型的预测偏离最优结果. 另一方面, 学习率过低也会降低模型的预测准确率, 这是因为较小的学习率会导致模型收敛速度变慢, 难以跳出局部最优值. 除此之外, 我们发现压缩率对模型预测精度的敏感度较高. 以 Sock Shop 数据集上的结果为例, 学习率 0.1 和学习率 0.001 时的预测准确率分别是 74.34% 与 77.72%, 二者仅相差了 3.38%, 但后者的压缩比是前者的 1.8 倍, 这在 Robot Shop 数据集上也有相似的结果. 为了解决以上问题, 我们以较大的学习率 0.001 作为 NCQT 的默认设置, 并且采用了自适应优化方法, 即优化器在训练和压缩过程动态调整学习率.

表 8 不同的学习率下的评估结果

学习率	Sock Shop		Robot Shop	
	准确率 (%)	文件大小 (MB)	准确率 (%)	文件大小 (MB)
0.1	74.34	25.39	82.81	33.46
0.01	76.82	16.50	84.19	22.53
0.001	77.72	13.99	85.11	18.03
0.000 1	77.73	13.99	85.15	17.92
0.000 01	77.61	14.11	84.79	18.42

为了提高模型的预测精度, NCQT 在压缩过程中利用压缩数据动态更新模型的参数. 为了验证该优化对压缩过程所产生的影响, 我们实现了两种不同的模型: 一种是压缩过程中模型参数冻结的静态模型, 另一种是在压缩过程中每次预测后根据预测结果更新参数的动态模型. 我们使用相同的模型参数设置和相同的训练后模型, 在两种模型上分别执行压缩过程, 并记录了它们各自的预测准确率、压缩后文件大小和压缩过程的时间开销, 实验结果如表 9 所示.

表 9 模型动态更新对压缩的影响

模型	Sock Shop			Robot Shop		
	准确率 (%)	文件大小 (MB)	压缩时间 (s)	准确率 (%)	文件大小 (MB)	压缩时间 (s)
静态模型	77.27	15.21	386	84.35	21.10	701
动态模型	77.73	13.98	1 031	85.18	17.90	1 877

从表 9 可以看出, 动态模型相较于静态模型压缩效果提升了 8%–18%, 压缩时间是后者的 2.7 倍. 实验结果表明, 动态模型相较静态模型需要更多的时间来压缩数据, 但是能够获得较高的预测准确率和更小的压缩文件大小. 因此, 为了平衡压缩效果与压缩效率, 我们通过设置更新时间步来限制模型参数更新的频率, 选取合理的时间步大小可以确保在可接受的时间范围内获得较好的压缩结果.

5 总 结

分布式追踪是分析和理解大型分布式系统的重要手段, 如何高效地压缩与查询追踪数据具有重要研究意义. 本文提出并实现了 NCQT, 一种基于神经网络的追踪压缩和查询方法. NCQT 可以有效识别追踪数据在内容与结构上的冗余, 并将自然语言文本转换为易于压缩与查询的中间表示. 然后, NCQT 使用神经网络模型对中间表示进行学习, 并通过算术编码将其压缩为更短的二进制序列. 此外, NCQT 对追踪数据进行分组处理, 在查询时过滤无关的单元, 可以在压缩数据上实现较为高效的查询. 我们在 4 个开源微服务系统的追踪数据集上进行了广泛的实验, 以评估 NCQT 的有效性. 实验结果表明, 与现有的通用数据压缩工具和神经网络压缩方法相比, NCQT 显著提高了压缩比. 对于追踪数据的查询, NCQT 在最佳情况下可以取得优于目前主流的追踪查询工具的实验结果.

本文提出的压缩方法是在单个节点上进行的, 压缩和查询效率受单机资源的限制. 在未来的工作中, 我们将研

究如何利用分布式集群对 NCQT 水平扩展, 进一步提高算法的并行度, 从而更高效地压缩和查询分布式追踪数据.

References:

[1] Yang Y, Li Y, Wu ZH. Survey of state-of-the-art distributed tracing technology. Ruan Jian Xue Bao/Journal of Software, 2020, 31(7): 2019–2039 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6047.htm [doi: 10.13328/j.cnki.jos.006047]

[2] Zhang ZZ, Ramanathan MK, Raj P, Parwal A, Sherwood T, Chabbi M. CRISP: Critical path analysis of large-scale microservice architectures. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 655–672.

[3] Zhang YZ, Isaacs R, Yue Y, Yang JC, Zhang L, Vigfusson Y. LatenSeer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In: Proc. of the 2023 ACM Symp. on Cloud Computing. Santa Cruz: ACM, 2023. 502–519. [doi: 10.1145/3620678.3624787]

[4] Zhang CX, Peng X, Sha CF, Zhang K, Fu ZQ, Wu XY, Lin QW, Zhang DM. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 623–634. [doi: 10.1145/3510003.3510180]

[5] Zhang K, Zhang CX, Peng X, Sha CF. PUTraceAD: Trace anomaly detection with partial labels based on GNN and PU learning. In: Proc. of the 33rd IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Charlotte: IEEE, 2022. 239–250. [doi: 10.1109/ISSRE55969.2022.00032]

[6] Zhou T, Zhang CX, Peng X, Yan ZH, Li PR, Liang JM, Zheng HB, Zheng WJ, Deng YT. TraceStream: Anomalous service localization based on trace stream clustering with online feedback. In: Proc. of the 34th IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Florence: IEEE, 2023. 601–611. [doi: 10.1109/ISSRE59848.2023.00033]

[7] Liu JS, Wang QY, Zhang SG, Hu LT, Da Silva D. Sora: A latency sensitive approach for microservice soft resource adaptation. In: Proc. of the 24th Int'l Middleware Conf. Bologna: ACM, 2023. 43–56. [doi: 10.1145/3590140.3592851]

[8] Huang LX, Zhu T. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In: Proc. of the 2021 ACM Symp. on Cloud Computing. Seattle: ACM, 2021. 76–91. [doi: 10.1145/3472883.3486994]

[9] Huang ZC, Chen PF, Yu GB, Chen HY. Transparent request tracing and sampling method for Java-based microservice system. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3167–3187 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6523.htm [doi: 10.13328/j.cnki.jos.006523]

[10] Peng X, Zhang CX, Zhao ZY, Isami A, Guo XF, Cui YN. Trace analysis based microservice architecture measurement. In: Proc. of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Singapore: ACM, 2022. 1589–1599. [doi: 10.1145/3540250.3558951]

[11] He SL, Feng BT, Li LQ, Zhang X, Kang Y, Lin QW, Rajmohan S, Zhang DM. STEAM: Observability-preserving trace sampling. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1750–1761. [doi: 10.1145/3611643.3613881]

[12] Gias AU, Gao YC, Sheldon M, Perusquía JA, O'Brien O, Casale G. SampleHST: Efficient on-the-fly selection of distributed traces. In: Proc. of the 2023 IEEE/IFIP Network Operations and Management Symp. (NOMS 2023). Miami: IEEE, 2023. 1–9. [doi: 10.1109/NOMS56928.2023.10154383]

[13] Zhou H, Chen M, Lin Q, Wang Y, She XB, Liu SF, Gu R, Ooi BC, Yang JF. Overload control for scaling WeChat microservices. In: Proc. of the 2018 ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 149–161. [doi: 10.1145/3267809.3267823]

[14] Annamalai M, Ravichandran K, Srinivas H, Zinkovsky I, Pan LN, Savor T, Nagle D, Stumm M. Sharding the shards: Managing datastore locality at scale with Akkio. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2018). Carlsbad: USENIX Association, 2018. 445–460.

[15] Las-Casas P, Mace J, Guedes D, Fonseca R. Weighted sampling of execution traces: Capturing more needles and less hay. In: Proc. of the 2018 ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 326–332. [doi: 10.1145/3267809.3267841]

[16] Las-Casas P, Papakerashvili G, Anand V, Mace J. Sifter: Scalable sampling for distributed traces, without feature engineering. In: Proc. of the 2019 ACM Symp. on Cloud Computing. Santa Cruz: ACM, 2019. 312–324. [doi: 10.1145/3357223.3362736]

[17] Huang ZC, Chen PF, Yu GB, Chen HY, Zheng ZB. Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems. In: Proc. of the 2021 IEEE Int'l Conf. on Web Services (ICWS). Chicago: IEEE, 2021. 436–446. [doi: 10.1109/ICWS53863.2021.00063]

[18] Free Software Foundation. GNU gzip. 2024. https://www.gnu.org/software/gzip/

[19] Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE Trans. on Information Theory, 1977, 23(3): 337–343. [doi: 10.1109/TIT.1977.1055714]

[20] Huffman D A. A method for the construction of minimum-redundancy codes. Proc. of the IRE, 1952, 40(9): 1098–1101. [doi: 10.1109/JRPROC.1952.273898]

[21] Grafana Labs. Grafana Tempo. 2024. https://grafana.com/oss/tempo/

[22] Jaeger. Jaeger: Open source, distributed tracing platform. 2024. https://www.jaegertracing.io/

[23] Witten IH, Neal RM, Cleary JG. Arithmetic coding for data compression. Communications of the ACM, 1987, 30(6): 520–540. [doi: 10.1145/214762.214771]

[24] Merhav N, Gutman M, Ziv J. On the estimation of the order of a Markov chain and universal data compression. IEEE Trans. on Information Theory, 1989, 35(5): 1014–1019. [doi: 10.1109/18.42210]

[25] Deutsch P. DEFLATE compressed data format specification version 1.3. 1996. [doi: 10.17487/RFC1951]

[26] Moffat A. Implementing the PPM data compression scheme. IEEE Trans. on Communications, 1990, 38(11): 1917–1921. [doi: 10.1109/26.61469]

[27] Goyal M, Tatwawadi K, Chandak S, Ochoa I. DZip: Improved general-purpose loss less compression based on novel neural network modeling. In: Proc. of the 2021 Data Compression Conf. (DCC). Snowbird: IEEE, 2021. 153–162. [doi: 10.1109/DCC50243.2021.00023]

[28] Schmidhuber J, Heil S. Sequential neural text compression. IEEE Trans. on Neural Networks, 1996, 7(1): 142–146. [doi: 10.1109/72.478398]

[29] Mahoney MV. Fast text compression with neural networks. In: Proc. of the 13th Int'l Florida Artificial Intelligence Research Society Conf. Orlando: AAAI Press, 2000. 230–234.

[30] Liu Q, Xu YL, Li Z. DecMac: A deep context model for high efficiency arithmetic coding. In: Proc. of the 2019 Int'l Conf. on Artificial Intelligence in Information and Communication (ICAIIC). Okinawa: IEEE, 2019. 438–443. [doi: 10.1109/ICAIIC.2019.8668843]

[31] Bellard F. Lossless data compression with neural networks. 2019. https://bellard.org/nncp/nncp.pdf

[32] Goyal M, Tatwawadi K, Chandak S, Ochoa I DeepZip: Lossless data compression using recurrent neural networks. arXiv:1811.08162, 2018.

[33] Mao Y, Cui YF, Kuo TW, Xue CJ. TRACE: A fast transformer-based general-purpose lossless compressor. In: Proc. of the 2022 ACM Web Conf. ACM, 2022. 1829–1838. [doi: 10.1145/3485447.3511987]

[34] Wei JY, Zhang GY, Wang Y, Liu ZW, Zhu ZY, Chen JC, Sun TT, Zhou Q. On the feasibility of parser-based log compression in large-scale cloud systems. In: Proc. of the 19th USENIX Conf. on File and Storage Technologies. USENIX Association, 2021. 249–262.

[35] Yao KD, Sayagh M, Shang WY, Hassan AE. Improving state-of-the-art compression techniques for log management tools. IEEE Trans. on Software Engineering, 2022, 48(8): 2748–2760. [doi: 10.1109/TSE.2021.3069958]

[36] Christensen R, Li FF. Adaptive log compression for massive log data. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 2013. 1283–1284. [doi: 10.1145/2463676.2465341]

[37] Lin H, Zhou JY, Yao B, Guo MY, Li J. Cowic: A column-wise independent compression for log stream analysis. In: Proc. of the 15th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing. Shenzhen: IEEE, 2015. 21–30. [doi: 10.1109/CCGrid.2015.45]

[38] Liu JY, Zhu JM, He SL, He PJ, Zheng ZB, Lyu MR. Logzip: Extracting hidden structures via iterative clustering for log compression. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). San Diego: IEEE, 2019. 863–873. [doi: 10.1109/ASE.2019.00085]

[39] Rodrigues K, Luo Y, Yuan D. CLP: Efficient and scalable search on compressed text logs. In: Proc. of the 15th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2021. 183–198.

[40] Ding HL, Yan S, Zhai J, Ma SQ. ELISE: A storage efficient logging system powered by redundancy reduction and representation learning. In: Proc. of the 30th USENIX Security Symp. USENIX Association, 2021. 3023–3040.

[41] Li XY, Zhang HY, Le VH, Chen PF. LogShrink: Effective log compression by leveraging commonality and variability of log data. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 23. [doi: 10.1145/3597503.3608129]

[42] Agarwal R, Khandelwal A, Stoica I. Succinct: Enabling queries on compressed data. In: Proc. of the 12th USENIX Symp. on Networked Systems Design and Implementation. Oakland: USENIX Association, 2015. 337–350.

[43] Pibiri GE, Petri M, Moffat A. Fast dictionary-based compression for inverted indexes. In: Proc. of the 12th ACM Int'l Conf. on Web Search and Data Mining. Melbourne: ACM, 2019. 6–14. [doi: 10.1145/3289600.3290962]

[44] Zhang F, Zhai JD, Shen XP, Wang DL, Chen Z, Mutlu O, Chen WG, Du XY. TADOC: Text analytics directly on compression. The VLDB Journal, 2021, 30(2): 163–188. [doi: 10.1007/s00778-020-00636-3]

[45] Wei JY, Zhang GY, Chen JC, Wang Y, Zheng WM, Sun TT, Wu JS, Jiang JW. LogGrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In: Proc. of the 18th European Conf. on Computer Systems. Rome: ACM, 2023. 452–468. [doi: 10.1145/3552326.3567484]

[46] Sigelman BH, Barroso LA, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, Shanbhag C. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report, dapper-2010-1, 2010. https://netman.aiops.org/~peidan/ANM2019/7.TraceAnomalyDetection/ReadingList/2010Google_Dapper.pdf

[47] MacQueen J. Some methods for classification and analysis of multivariate observations. In: Proc. of the 5th Berkeley Symp. on Mathematical Statistics and Probability. Berkeley: University of California, 1967. 281–297.

[48] Subakan C, Ravanelli M, Cornell S, Bronzi M, Zhong JY. Attention is all you need in speech separation. In: Proc. of the 2021 IEEE Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP 2021). Toronto: IEEE, 2021. 21–25. [doi: 10.1109/ICASSP39728.2021.9413901]

[49] Zhou X, Peng X, Xie T, Sun J, Ji C, Li WH, Ding D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. on Software Engineering, 2021, 47(2): 243–260. [doi: 10.1109/tse.2018.2887384]

[50] Zhou X, Peng X, Xie T, Sun J, Xu CJ, Ji C, Zhao WY. Benchmarking microservice systems for software engineering research. In: Proc. of the 40th Int'l Conf. on Software Engineering: Companion Proc. Gothenburg: ACM, 2018. 323–324. [doi: 10.1145/3183440.3194991]

[51] Moffat A, Neal RM, Witten IH. Arithmetic coding revisited. ACM Trans. on Information Systems, 1998, 16(3): 256–294. [doi: 10.1145/290159.290162]

[52] Golmant N, Vemuri N, Yao ZW, Feinberg V, Gholami A, Rothauge K, Mahoney MW, Gonzalez J. On the computational inefficiency of large batch sizes for stochastic gradient descent. arXiv:1811.12941, 2018.

附中文参考文献:

[1] 杨勇, 李影, 吴中海. 分布式追踪技术综述. 软件学报, 2020, 31(7): 2019–2039. http://www.jos.org.cn/1000-9825/6047.htm [doi: 10.13328/j.cnki.jos.006047]

[9] 黄梓程, 陈鹏飞, 余广坝, 陈泓仰. 面向 Java 微服务系统的透明请求追踪及采样方法. 软件学报, 2023, 34(7): 3167–3187. http://www.jos.org.cn/1000-9825/6523.htm [doi: 10.13328/j.cnki.jos.006523]

王尚(1999－), 男, 硕士生, 主要研究领域为云原生, 智能化运维.

彭鑫(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能化软件开发, 云原生, 智能化运维, 泛在计算软件系统.

张晨曦(1992－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为智能化运维, 云原生, 软件测试.