数据库时序数据压缩技术

王思成

Jun 15, 2026

随着物联网、金融交易、运维监控和日志分析等场景的迅猛发展，时间序列数据的规模正以指数级速度膨胀。传统的通用存储方案难以同时满足高吞吐写入与低延迟查询的双重要求，因此专门针对时序数据的压缩算法与存储引擎应运而生。有效的压缩不仅能显著降低存储成本，还能通过减少 I/O 量和提高缓存命中率来改善整体查询性能。接下来，本文将系统梳理时序数据的特征、主流压缩算法及其原理、典型系统实践、性能权衡以及工程落地建议，帮助读者建立完整的知识体系。

1　时序数据的特征与压缩难点

时序数据通常由三元组构成：精确的时间戳、携带业务含义的指标或标签，以及对应的数值或文本内容。数据在时间轴上往往呈现单调递增、阶跃变化、周期波动或稀疏分布等特征，同时可能混杂测量噪声与异常尖峰。写入模式以追加为主，查询则集中在时间范围过滤和聚合运算上，这就要求压缩格式既能高效编码连续值，又能在不完全解压的情况下支持跳过与下推计算。乱序写入、数据过期淘汰、模式演进以及多租户隔离等约束进一步增加了压缩方案的设计复杂度。

2　压缩算法分类与原理

2.1　通用无损压缩

字典编码与熵编码是两类最基础的无损压缩手段。字典编码通过构建重复子串的映射表，用短索引代替长字符串，从而减少冗余；Zstandard 与 LZ4 均采用此思路，但前者通过更复杂的匹配窗口换取更高压缩率，后者则优先保证极低的解压延迟。熵编码则基于符号出现概率分配变长码字，Huffman 树与非对称数字系统（ANS）是两种典型实现。通用算法对任何字节流均有效，但未能充分利用时序数据的单调性与局部相关性，因此在高吞吐场景中往往作为第二层封装，与专用编码组合使用。

2.2　时间戳专用编码

时间戳序列具有严格单调或近似单调的特性，Delta-of-Delta 编码正是利用这一特性。假设相邻时间戳之差为 $\Delta_i = t_i - t_{i-1}$，则二阶差分为 $\Delta^2_i = \Delta_i - \Delta_{i-1}$。由于采样周期通常稳定，$\Delta^2_i$ 多为零或极小整数，可用变长编码紧凑表示。Facebook Gorilla 与 InfluxDB TSM 均采用此方法，并配合简单的符号位标记来区分正负与零值。变长整数编码如 LEB128 将数值拆分为 7 位有效位加 1 位延续标志，可在单字节内表示 0 – 127 的整数，超出范围则自动扩展字节数。两种编码结合后，时间戳列的存储开销可降低至原始长整型的十分之一左右。

2.3 数值与浮点专用编码

浮点数值在相邻采样点间往往只在低位比特发生变化，XOR 差分编码正是捕捉这一规律。将当前值与前一个值进行按位异或，得到的结果中前导零与后缀零的数量可用 3 – 6 比特描述，中间有效位则按需存储。Gorilla 论文给出的伪代码如下：

```
func encodeXOR(prev, curr uint64) []byte {
   xor := prev ^ curr
   if xor == 0 {
      return []byte{0x00} // 连续重复值仅需 1 比特标记
   }
   leading := bits.LeadingZeros64(xor)
   trailing := bits.TrailingZeros64(xor)
   significant := 64 - leading - trailing
   // 用 5 比特存 leading，6 比特存 significant，再写入有效负载
   ...
}
```

该算法在保持全精度无损的前提下，将 64 位浮点压缩至平均 1 – 2 字节。另一种思路是将浮点拆分为符号位、指数位与尾数位，分别采用游程编码或字典编码。BTrDB 将数据分块后先做行程长度编码，再对块内残差进行二次压缩，在高精度场景下表现出色。线性或二阶多项式拟合则通过少量系数描述一段连续曲线，仅存储残差，适用于变化平缓的物理量监测。

2.4 列式位图与索引压缩

列式存储天然适合位图与行程编码。Roaring Bitmap 将 32 位整数划分为高 16 位容器与低 16 位数组或位图，根据基数自适应切换存储形式，既保持了 O(1) 的随机访问，又大幅降低了内存占用。行程编码（RLE）对重复值连续出现的列效果显著，只需存储值本身与重复次数两项信息。字典序前缀编码（FOR）与 Delta 编码则在 Parquet 文件格式中广泛使用：先将整列减去最小值，再用定宽或变宽整数存储，从而消除高位冗余。

2.5 近似与有损压缩

当精度需求可松弛时，近似算法能进一步降低存储。分位数摘要（如 T-Digest）将原始数据映射到固定数量的桶中，用桶中心与计数近似分布，从而支持任意分位数查询却仅占用 KB 级空间。Facebook 后续研究尝试用分段线性回归拟合时序曲线，只保留断点坐标与斜率，重建时通过线性插值恢复，压缩率可达 100:1 以上，但需权衡重建误差对下游分析的影响。异常值剥离策略先检测并单独存储尖峰，再对主体序列强力压缩，可在保证异常不丢失的前提下提升整体压缩率。

2.6 混合与自适应策略

单一算法难以兼顾所有数据模式，因此自适应框架成为主流。TimescaleDB 将 hypertable 按时间与空间切分为若干 segment，在 segment 内部根据列的基数、重复度与单调性动态选择编码方式。Apache IoTDB 则

在 Chunk 与 Page 两级进行算法竞争：Chunk 头记录候选算法列表，写入线程实时采样前若干 Page 的压缩率与耗时，选出最优者并更新 Chunk 元数据。块大小自适应同样关键，过小的块导致元数据膨胀，过大的块则增加 Compaction 时的重写开销。

3 典型开源与商业系统实践

Facebook 的 Gorilla 与 Beringei 最早系统性地将 Delta-of-Delta 与 XOR 编码落地到生产环境。Gorilla 通过内存中的循环缓冲区实现实时压缩，Beringei 则在持久化层增加后台合并线程，进一步提升冷数据压缩率。InfluxDB 的 TSM 引擎将时间戳、字段值与序列号分别组织成独立的列式块，每块内部先用 Delta-of-Delta 与 XOR 编码，再外层套用 Snappy，兼顾了压缩率与查询速度。TimescaleDB 基于 PostgreSQL 扩展 hypertable，将数据按时间分区并独立压缩，每个分区内再按标签分组形成 segment，从而在多租户隔离与压缩效率之间取得平衡。Apache IoTDB 设计了 ChunkGroup-Page 两级结构，ChunkGroup 对应一个设备的一段时间窗口，内部 Page 按列存储并独立编码，支持快速跳过无关设备的数据。Prometheus TSDB 将最近两个小时的活跃数据保留在内存 Head Block 中，定期 Flush 成不可变块并执行块内再压缩；WAL 则以追加方式记录原始样本，确保崩溃恢复。ClickHouse 将列式存储与物化视图相结合，物化视图预先聚合常用指标，既减少了存储量，又加速了聚合查询。TDengine 通过超级表与子表的分区分片机制，将同一设备的数据物理集中，极大提升了 RLE 与字典编码的效率。VictoriaMetrics 在块头中记录统计元数据与稀疏索引，使查询引擎能在不解压数据块的情况下完成时间范围过滤与标签匹配。

4 压缩与查询性能的权衡

压缩率与解压延迟是一对经典矛盾。高压缩率算法通常需要更复杂的上下文建模，从而增加 CPU 开销；反之，轻量级算法如 LZ4 在解压时仅需查表与拷贝，可在亚毫秒级完成。现代分析型引擎普遍要求在压缩块内部直接执行过滤、聚合或跳过操作，这就要求压缩格式暴露足够的元数据，例如块内最小最大值、SUM/COUNT 草图以及稀疏索引。SIMD 指令集与 GPU 可显著加速解压，Zstandard 的 AVX2 实现已将解压吞吐提升至 5 GB/s 以上。缓存友好性方面，列式布局在聚合场景下访问局部性更好，而行式布局更适合点查与宽表投影，混合布局如 Apache Arrow 的 RecordBatch 可在二者间动态切换。

5 工程实践建议

在数据流转路径上，采集端可进行轻量级压缩以降低网络带宽，数据库端再做二次强压缩以优化存储。冷热分层策略将最近若干小时或天的数据保留在热存储并采用低压缩率算法，历史数据则迁移至对象存储并启用高压缩率算法。乱序写入会破坏已压缩块的局部相关性，因此多数系统在后台定期执行 Compaction，将乱序数据重新排序与合并。监控体系需跟踪压缩率、P99 解压延迟与 CPU 使用率三项核心指标，及时发现异常。加密与压缩可协同工作，但需注意压缩后再加密会失去字典与熵编码的优势，因此通常先压缩再加密；校验和应覆盖压缩块与元数据，避免静默数据损坏；备份策略则需记录压缩算法版本，确保跨版本恢复兼容。

6 未来趋势

硬件加速是下一阶段的重要方向，FPGA 与 ASIC 可将解压逻辑固化到芯片上，将延迟降至纳秒级。机器学习驱动的自适应压缩利用预测模型生成残差序列，再对残差进行传统编码，可在非平稳序列上获得额外增益。云原生存算分离架构下，对象存储的压缩格式需支持并行 Range-Get 与服务器端过滤，Apache Parquet 与 ORC 已在这方面持续演进。标准化工作同样关键，Apache Arrow 定义了统一的压缩扩展接口，OpenTelemetry 协议则在采集层引入可选的 zstd 压缩选项，力求在生态各层实现互操作。

时序数据压缩没有放之四海而皆准的银弹，必须根据数据温度、访问模式与精度需求灵活组合算法。持续演进要求算法、存储引擎与硬件三者协同迭代，只有在工程实践中不断测量、调优与验证，才能在存储成本与查询性能之间找到最优平衡。