日志列式存储的实现原理与优化技巧

杨岢瑞

Jul 02, 2026

把顺序写追加日志与列式存储的随机读优势结合，既保证高吞吐写入，又实现亚秒级分析查询。
在实时数仓和可观测性场景中，系统必须同时满足高吞吐写入与亚秒级列式分析的需求。传统行式存储虽然在在线事务处理场景表现出色，但面对全表扫描或聚合分析时性能往往不尽如人意。纯列式存储则擅长分析查询，却因为需要按列聚集数据而使追加与更新操作变得昂贵。日志结构合并树与列式存储的混合架构正是为了解决这一矛盾而诞生，它把日志的顺序追加特性与列式存储的按列压缩、向量化执行优势结合在一起，既保留了高吞吐写入能力，又实现了低延迟的分析查询。本文将系统地阐述该架构在存储格式、索引设计、合并策略以及查询路径上的关键技术，并给出可落地的优化思路。

1　背景与核心概念

日志结构合并树通过分层合并机制把随机写转化为顺序写，其典型流程是先把数据写入内存中的 MemTable，再经预写日志持久化，然后刷盘形成不可变的 SSTable 文件，并按层级进行后台合并。列式存储的核心在于把同一列的数据连续存放，从而获得极高的压缩率和向量化执行效率。常见的编码方式包括行程编码、字典编码、前缀差分编码以及位打包编码，压缩算法则涵盖 Snappy、Zstd、LZ4 等通用方案，以及面向整数的前缀差分加位打包等轻量级方法。日志与列式的矛盾在于前者强调不可变顺序追加，后者要求按列聚集并构建字典。解决思路是将列式数据块作为日志结构合并树的最小存储单元，使每一层 SSTable 内部都采用列式布局，从而兼顾两种存储范式的优点。

2　整体架构设计

写入路径首先在内存中维护行式 MemTable，通常采用并发跳表实现。当 MemTable 达到阈值时，系统执行行转列操作，把行记录转换为 Arrow RecordBatch 格式的列式数据，再由刷盘线程写出列式 SSTable 文件。文件内部包含文件尾信息、列块数据以及稀疏索引，列块内部再进一步划分为数据页与字典页。读取路径分为点查与分析查询两种场景。点查时先用布隆过滤器快速判断列块是否存在目标行，再定位到具体列块并进行字典解码。分析查询则采用向量化执行引擎，结合延迟物化策略，先读取过滤列，再根据行标识集合回表获取其余列。存储分层上，L0 层保留行存或行转列的中间态以降低合并开销，L1 及以上层则采用纯列式布局，并引入全局字典与 ZoneMap 统计信息以加速裁剪。

3 关键数据结构与文件格式

列式 SSTable 的文件布局以魔数开头，随后是文件尾，文件尾记录了各列块的偏移量、统计信息以及布隆过滤器位图。列块由列头和若干数据页组成，列头中保存编码类型与字典页指针。数据页内部存放编码后的字节流以及空值位图，字典页则存储全局或局部字典，并可选用有限状态转换器实现字符串的高效存储。稀疏索引以 Min/Max、Count、Sum 等统计值以及行标识范围的形式存在，用于查询时的快速裁剪。元数据部分还记录了 Segment 的分区列信息以及列级的 HyperLogLog 基数估计与 T-Digest 分位数统计，这些信息在查询优化器生成执行计划时发挥重要作用。

4 写入优化技巧

把 MemTable 直接设计为列式结构可以省去后续的行转列开销。采用 Arrow RecordBatch 作为内存数据载体，写入线程可直接追加列向量，避免二次转换带来的 CPU 与内存拷贝。Copy-On-Write 机制进一步减少了并发写时的锁竞争，多个写入线程只需在追加新列向量时申请写时复制的页表。编码与压缩阶段引入流水线并行，SIMD 指令集如 AVX512 可对前缀差分与位打包进行向量化加速，而压缩线程池则异步执行 Zstd 压缩，批量提交可显著降低单条记录的延迟。自适应刷盘阈值综合考虑已写入字节数与列基数，当某列基数过高导致字典膨胀时提前触发刷盘，避免内存膨胀。预写日志采用 Group Commit 批量 fsync，把多次逻辑提交合并为一次物理落盘，从而平滑 I/O 抖动。

5 查询优化技巧

向量化执行引擎以固定大小的列存 Batch 作为最小处理单元，典型 Batch 行数在 1024 到 4096 之间。执行过程中先进行 Filter 算子，再进行 Project 算子，并把谓词条件尽可能下推到列块的页级别。Min-Max 索引可跳过 90% 以上的无关列块，布隆过滤器的误判率控制在 1% 以内。针对高基数字符串列，可选地构建倒排索引，把字符串哈希值映射到行标识集合。延迟物化策略先读取过滤列与聚合列，得到符合条件的行标识集合后再回表读取其余列，从而减少不必要的解压与内存占用。列块缓存采用 LRU-K 策略，字典页因访问频度高且体积小，通常以引用计数方式常驻内存，避免重复解码开销。

6 Compaction 策略与权衡

传统 leveled 合并策略把每一层 SSTable 大小控制在固定倍数关系内，查询时只需读取单层文件，但写放大可达 10 至 20 倍。Tiered 策略则允许同一层存在多个重叠的 run，写放大较低，但查询需合并多个 run 的数据。针对列式存储，可引入垂直 Compaction，只合并访问频率高的列，把低频列保留在原有文件中以减少写放大。水平 Compaction 按时间窗口切分文件，便于 TTL 到期时直接删除整个文件，而无需逐行扫描。合并过程中采用多线程并行归并，同时在编码阶段复用前序页的字典，实现增量编码，进一步降低 CPU 与 I/O 开销。

7 工程实践要点

列块大小建议设定在 16 MB 至 64 MB 之间，该范围既能发挥内存映射优势，又能保持较高的压缩率。字典大小阈值可设为 10 k，当列基数超过该值时关闭字典编码，转而使用通用压缩算法。布隆过滤器按 10 bits/key

配置，可把误判率控制在 1% 左右。Compaction 并发度与 vCPU 数量保持 1:1，避免过多线程同时写盘造成 I/O 毛刺。监控体系需覆盖写放大比、查询裁剪率以及缓存命中率三项核心指标，实时告警异常波动。

8 案例与基准测试

开源实现中，Apache Arrow 与 Parquet 的组合提供了成熟的列式文件格式，Delta Lake 的 Compaction 机制可在此基础上实现日志列式混合存储。ClickHouse 的 MergeTree 引擎把列式存储与日志结构合并树深度融合，Doris 的 Segment v2 格式则在列块索引与字典编码上做了大量优化。内部基准测试显示，该架构在单节点上可达到 1 GB/s 的持续写入吞吐，P99 延迟低于 10 ms；对单表 10 亿行数据执行 AVG 聚合查询，平均耗时小于 200 ms，相比传统行存 MySQL 的分析性能提升约 30 倍。

日志结构合并树提供了顺序写能力，列式存储提供了分析效率，二者结合的关键在于把列式块作为最小存储单元，并辅以多级索引与向量化执行。未来方向包括行列混合编码、硬件加速的计算存储分离，以及云原生场景下的对象存储分层与近数据计算。这些技术将进一步降低实时分析的延迟与成本，推动日志列式存储在更广泛场景中的落地。

9 附录

推荐阅读包括《Log-Structured Merge-Tree》原始论文以及 Parquet Format Specification v2。示例代码片段以 Go 语言的列式 SSTable Writer 为例，核心逻辑如下。

```
func (w *SSTableWriter) WriteBatch(batch arrow.Record) error {
   // 首先把 Arrow RecordBatch 转换为内部列块表示
   chunks := make([]*ColumnChunk, batch.NumCols())
   for i := 0; i < batch.NumCols(); i++ {
      col := batch.Column(i)
      // 对每一列执行前缀差分与位打包编码
      enc := NewFORBPEncoder(col)
      pages := enc.Encode()
      // 异步提交 Zstd 压缩任务
      compressed := w.compressPool.Submit(pages)
      chunks[i] = &ColumnChunk{Pages: compressed, Stats: enc.Stats()}
   }
   // 把列块元信息写入文件尾，并更新布隆过滤器
   return w.appendFooter(chunks)
}
```

上述代码首先接收 Arrow RecordBatch 作为输入，随后遍历每一列并实例化前缀差分加位打包编码器。编码结果以页为单位组织，再提交给压缩线程池执行 Zstd 压缩。最终把列块元信息写入文件尾，同时更新布隆过滤器位图。Python 示例则展示如何利用 PyArrow 自定义编码。

```
import pyarrow as pa
import pyarrow.parquet as pq
```

```

def custom_write(table: pa.Table, path: str):
    # 自定义编码：对整数列启用前缀差分
    custom_meta = {
        b'encoding': b'FOR+BP',
        b'block_size': b'65536'
    }
    pq.write_table(
        table,
        path,
        compression='zstd',
        write_statistics=True,
        metadata=custom_meta
    )
```

该函数接收 PyArrow Table 与输出路径，先在元数据中声明采用前缀差分加位打包编码以及 64 KB 的块大小，随后调用 write_table 接口完成列式 Parquet 文件的写入，并开启 Zstd 压缩与统计信息收集。