数据库存储引擎设计原理

黄京

Nov 04, 2026

存储引擎是数据库管理系统中最核心的部件，它决定了数据如何被组织、如何被检索、如何在并发访问下保持一致性，以及在系统崩溃后如何恢复。理解存储引擎不仅能帮助我们更好地调优现有数据库，更能让我们在面对海量数据或特殊场景时做出正确的架构决策。

1　存储引擎在 DBMS 中的定位

在典型的数据库架构中，存储引擎位于执行引擎之下，SQL 解析与优化后的逻辑操作最终都由存储引擎完成物理读写。它需要同时满足持久化、事务原子性、隔离性和持久性等 ACID 特性，同时还要在读写放大、空间占用与查询延迟之间寻找平衡。换句话说，存储引擎既是「磁盘管理器」，又是「并发调度器」与「崩溃恢复器」。

2　数据组织模型的选择

行式存储把一条记录的所有列连续存放，适合 OLTP 场景中频繁的点查与更新。InnoDB 与 PostgreSQL 的堆表都采用这一模型，其优势在于一次 I/O 即可拿到整行，但当只需要少数列做聚合时，会产生大量无效 I/O。列式存储则把同一列的数据连续存放，ClickHouse 与 Parquet 都以此见长，极高的压缩率与向量化执行让它在 OLAP 场景下性能碾压。然而，列存更新代价高昂，通常只能追加写入。LSM-Tree 把更新先写入内存 MemTable，再周期性刷盘生成不可变 SSTable，通过后台 Compaction 合并文件，既获得了顺序写盘的高吞吐，又能用 Bloom Filter 把读放大控制在可接受范围。RocksDB、LevelDB 及 HBase 均采用这一模型。

3　B+ 树索引的实现细节

B+ 树是关系型数据库中最常见的索引结构。它的每个节点都存储若干键值对和子节点指针，内部节点仅保存索引而叶子节点存放完整记录或指向记录的指针。插入时若目标叶子节点已满，会执行分裂操作：先把节点中点位置的键提升到父节点，再把原节点拆成左右两个新节点。分裂会逐级向上传播，直到根节点也需要分裂，此时整棵树高度增加一层。删除则触发合并或借位，以维持「至少半满」的填充因子，避免频繁的磁盘随机写。

在 InnoDB 中，表数据本身就是一棵 B+ 树，称为聚簇索引。主键顺序即物理存储顺序，因此范围查询可以直接顺序扫描叶子链表，极大减少随机 I/O。辅助索引则只存放主键值，二次查询再回表拿到完整行。

4 WAL 与崩溃恢复

为了把随机写转化为顺序写，同时保证事务持久性，存储引擎普遍采用预写日志（WAL）。每条修改在落盘前必须先写入日志文件，并携带单调递增的日志序列号（LSN）。当系统崩溃重启时，恢复过程分为三阶段：Analysis 根据 Checkpoint 定位脏页集合；Redo 重放所有 LSN 大于 Checkpoint 的日志，使数据回到崩溃前状态；Undo 则回滚未提交的事务，保证原子性。ARIES 算法把这三个阶段解耦，使得恢复时间与日志量呈线性关系，而非与数据量相关。

5 MVCC 与并发控制

多版本并发控制让读操作不必阻塞写操作。InnoDB 为每一行维护隐藏的两列：创建版本号与删除版本号。每个事务在启动时会生成一个 ReadView，里面记录了「当前系统中活跃事务 ID 列表」以及「当前最大事务 ID」。当事务读取一行记录时，会用 ReadView 判断该版本是否对本事务可见：若创建版本号大于 ReadView 中的最大事务 ID，或创建版本号在活跃列表中，则该版本不可见，需要沿 Undo Log 链条继续寻找更早版本。可见性判断仅涉及比较操作，因此在高并发读写混合场景下性能优于锁机制。

6 缓冲池与 I/O 策略

磁盘 I/O 是数据库最主要的瓶颈。缓冲池把磁盘页缓存到内存，并使用改进的 LRU 策略管理。InnoDB 把缓冲池分成若干个「缓冲池实例」以减少锁竞争，同时引入「老生代」与「新生代」两个链表，新读入的页先放老生代，只有经过再次访问才会晋升到新生代，避免偶发的大表扫描把热数据全部淘汰。刷脏页时采用「自适应刷盘」算法，根据 redo 日志生成速度动态调整刷盘速率，避免 Checkpoint 时瞬间 I/O 风暴。

Direct I/O 与 mmap 是两种绕过操作系统缓存的方案。前者让数据库自己管理页缓存，消除双份缓存；后者把文件映射到虚拟地址空间，把文件读写转化为内存访问，适合读多写少场景。

7 典型引擎对比

MyISAM 只支持表级锁，没有 WAL，崩溃后需 REPAIR TABLE，适合只读或低并发写入。InnoDB 引入行级锁、MVCC、Doublewrite Buffer 与自适应哈希，全面支持 ACID，成为 MySQL 默认引擎。RocksDB 针对 SSD 做了优化：Column Family 让不同负载共享同一日志；分层 Compaction 把文件按重叠键范围组织，降低读放大；BlobDB 把大 Value 单独存放，减少 Compaction 写放大。

TiDB、OceanBase 与 CockroachDB 都是分布式 SQL 数据库，但底层存储引擎差异明显：TiDB 使用 TiKV（基于 RocksDB），数据按 Region 范围划分；OceanBase 采用 Paxos 复制的 LSM-Tree，并支持多租户资源隔离；CockroachDB 则基于 Raft 与 RocksDB 实现全局一致性与自动再平衡。

8 性能调优要点

innodb_buffer_pool_size 通常建议设为物理内存的 50% – 70%，但若存在大量大表顺序扫描，过大的缓冲池反而会增加 LRU 链表维护开销。rocksdb_max_write_buffer_number 控制内存中 MemTable 数量，值太小会频繁触发 Flush，太大会导致内存占用过高。监控指标方面，Buffer Pool 命中率低于 99% 说明工作

集已超过内存；WAL 平均刷盘延迟超过 10 ms 则需要检查磁盘队列深度或开启 Group Commit。

9 未来方向

持久化内存（PMem）让字节寻址持久化成为可能，存储引擎可把 B+ 树节点直接放在 PMem 上，省去 WAL 与 Checkpoint。云原生环境下，存储与计算分离后，共享存储层需要支持快照、克隆与跨可用区复制，同时还要提供与计算节点带宽匹配的 I/O 能力。近数据计算则把过滤、聚合等简单算子下推到存储节点，减少网络传输。AI 辅助索引能够根据历史 workload 自动创建或删除索引，并动态调整 Compaction 策略，让系统始终处于最优状态。

存储引擎的设计本质是在数据结构、并发模型与 I/O 策略三者间不断权衡。掌握这些原理，我们才能在选型、调优乃至自研数据库时做出有根据的决策。