SQL 调优与索引设计

王思成

Aug 03, 2026

在现代应用系统中，SQL 性能优化直接决定了企业的运营成本和用户体验。当查询响应时间从百毫秒级上升到秒级时，用户流失率可能增加 30% 以上；当数据库 CPU 持续满载时，横向扩展的成本会呈指数级上升。索引作为 SQL 调优中最常见且投入产出比最高的手段，能够以 1% 的存储代价换取 90% 的查询加速。本文面向 DBA、后端及全栈工程师、架构师，系统梳理从索引原理到生产实践的全链路知识。

1　基础概念与原理

关系型数据库普遍采用 B+Tree 作为默认索引结构。B+Tree 的所有叶子节点通过双向链表相连，内部节点仅存储键值和子节点指针，叶子节点同时存放键值和数据指针或整行记录。这种设计使得范围查询可以转化为顺序 I/O，而非随机 I/O。Hash 索引仅支持等值查询且不支持范围扫描，在 MySQL 的 Memory 引擎中仍有应用。Bitmap 索引适合低基数列，通过位图向量实现快速 AND/OR 操作，常用于数据仓库。GiST 和 GIN 则为 PostgreSQL 提供的扩展索引类型，前者支持几何、文本等复杂类型，后者专为倒排索引和数组优化。

聚簇索引决定了数据行的物理存储顺序，InnoDB 的主键即为聚簇索引，二级索引的叶子节点存放的是主键值而非行指针，因此通过二级索引查询后通常需要一次「回表」操作才能获取完整记录。非聚簇索引则将数据与索引分离存储，SQL Server 的堆表和 PostgreSQL 的 heap 均属于此类。覆盖索引是指索引中包含查询所需全部列，从而避免回表，例如在 (user_id, created_at) 索引上执行 SELECT user_id, created_at WHERE user_id = 1 的查询即可直接命中覆盖索引。

复合索引遵循最左前缀原则，索引 (a, b, c) 可服务于 a、a+b、a+b+c 三种查询模式，但无法服务于单独的 b 或 c 查询。函数索引允许在表达式上建立索引，例如 PostgreSQL 的 CREATE INDEX idx ON t ((lower(email))) 可加速不区分大小写的邮箱查询。部分索引通过 WHERE 子句限制索引范围，例如只为 status = 'active' 的记录建立索引，显著降低索引体积。

索引会同时放大读写两类开销。每次写入不仅要修改数据页，还要同步更新所有相关索引，导致写放大；查询时若索引选择性差，可能引发全表扫描或大量回表随机 I/O。空间占用方面，主键长度每增加 8 字节，二级索引也会相应增长，因此在高并发写入场景下需谨慎评估索引数量。

执行计划的成本模型通常以逻辑读次数和 CPU 周期为单位。MySQL 的 EXPLAIN 结果中，type 字段反映访问类型，rows 字段估算扫描行数，filtered 字段表示条件过滤比例。PostgreSQL 的 EXPLAIN (ANALYZE, BUFFERS) 可进一步显示实际执行时间和缓冲区命中情况，帮助判断索引是否真正被有效利用。

2 索引设计黄金法则

高选择性列应放在复合索引前列。选择性定义为 distinct_values / total_rows，接近 1 的列区分度最高。例如性别列的选择性通常低于 0.1，而 UUID 主键的选择性接近 1。将高选择性列前置可使索引树快速收敛，减少后续扫描范围。

最左前缀匹配要求查询条件必须包含索引最左侧连续列。索引 (user_id, order_status, created_at) 可支持 WHERE user_id = 1 AND order_status = 'paid' 的等值查询，也支持 WHERE user_id = 1 AND order_status = 'paid' AND created_at > '2024-01-01' 的范围查询，但无法支持 WHERE order_status = 'paid' 的单独查询。

在索引列上使用函数或类型转换会导致索引失效。例如 WHERE YEAR(created_at) = 2024 无法使用 (created_at) 索引，因为索引存储的是原始值而非函数结果。应改写为 WHERE created_at >= '2024-01-01' AND created_at < '2025-01-01'，使查询条件与索引存储格式一致。

索引列顺序需与查询模式匹配。对于 WHERE user_id = 1 ORDER BY created_at 的场景，索引 (user_id, created_at) 可同时满足等值过滤和排序，避免 filesort。对于 WHERE user_id = 1 AND created_at > '2024-01-01' ORDER BY created_at 的范围查询，同样需要 created_at 放在 user_id 之后，以确保范围条件能够利用索引顺序。

索引数量和宽度需严格控制。单表索引建议不超过 5 个，单索引列数不超过 3 个。过宽索引不仅占用更多存储空间，还会增加写入时的页分裂概率。唯一索引与普通索引的区别在于前者需要额外维护唯一性约束，且在冲突检测时可能引发死锁。主键天然具有唯一索引属性，且作为聚簇索引的组织键，不应频繁更新。

3 典型场景的索引策略

等值查询 (user_id = 123) 可使用 ref 或 eq_ref 访问类型，索引 (user_id) 即可满足。若涉及 IN (1,2,3) 多值匹配，MySQL 会将 IN 列表排序后做 range 扫描，索引 (user_id) 仍有效，但列表长度超过 1000 时需警惕性能下降。

范围查询 (created_at > '2024-01-01') 要求范围列放在索引末尾。索引 (user_id, created_at) 支持 WHERE user_id = 1 AND created_at > '2024-01-01'，却不支持 WHERE created_at > '2024-01-01' AND user_id = 1 后的范围裁剪，因为 user_id 的等值条件必须出现在范围条件之前。LIKE 'abc%' 前缀匹配可退化为范围查询，而 LIKE '%abc' 则必须全表扫描或借助全文索引。

多表 JOIN 的性能关键在于连接列是否建立索引。对于 users JOIN orders ON users.id = orders.user_id 的查询，需在 orders.user_id 上建立索引。若连接顺序为小表驱动大表，则小表的全表扫描成本更低；反之则需在大表连接列上建立索引以支持 index lookup。

ORDER BY 优化要求排序列与索引顺序一致。索引 (user_id, created_at DESC) 可直接支持 ORDER BY user_id, created_at DESC，避免 filesort。若排序方向不一致，例如 ORDER BY user_id, created_at ASC，则需单独建立 (user_id, created_at ASC) 索引，MySQL 8.0 已支持降序索引。

深分页问题源于 LIMIT 100000, 10 需要先扫描 100010 行再丢弃前 100000 行。解决方案包括延迟关联和游标分页。延迟关联先用覆盖索引定位主键，再回表查询完整行：SELECT * FROM orders o JOIN (SELECT id FROM orders WHERE user_id = 1 ORDER BY created_at LIMIT 100000, 10) tmp ON o.id = tmp.id。游标分页则通过 WHERE created_at > last_created_at LIMIT 10 实现无 offset 的稳定性能。

全文检索需使用专门的倒排索引结构。MySQL 的 FULLTEXT 索引支持自然语言和布尔模式查询，底层采用倒排文件存储词项到文档 ID 的映射。PostgreSQL 的 tsvector 类型配合 GIN 索引可实现中文分词和短语查询，需配合 pg_trgm 扩展支持模糊匹配。

JSON/数组/地理位置等半结构化数据索引依赖数据库特定扩展。PostgreSQL 的 JSONB 类型可使用 GIN 索引加速 @> 和 ? 操作符查询，例如 CREATE INDEX idx ON t USING GIN (data jsonb_path_ops)。MySQL 8.0 的多值索引支持 JSON 数组元素的单独索引，适用于标签类查询场景。地理位置索引通常采用 R-Tree 或 GiST 结构，支持 ST_DWithin 等空间关系查询。

4 SQL 语句层面的调优技巧

SELECT * 会导致回表次数增加且无法利用覆盖索引。应明确列出所需字段，例如 SELECT user_id, created_at 而非 SELECT *，使优化器有机会选择 (user_id, created_at) 覆盖索引。

绑定变量可避免硬解析开销。字面量查询 SELECT * FROM orders WHERE user_id = 123 每次执行需生成新执行计划，而绑定变量 SELECT * FROM orders WHERE user_id = ? 可复用计划，减少 CPU 消耗。但绑定变量也可能导致参数嗅探问题，需结合执行计划缓存策略权衡。

OR 条件常导致索引失效。WHERE user_id = 1 OR order_status = 'paid' 无法使用单一索引，需改写为 UNION：SELECT * FROM orders WHERE user_id = 1 UNION SELECT * FROM orders WHERE order_status = 'paid' AND user_id <> 1。改写后每个分支可独立使用索引，但需注意去重开销。

EXISTS、IN、JOIN 的性能差异取决于子查询返回行数。IN 子查询若返回少量值，优化器可能改写为半连接；若返回大量值，则可能转为反连接或物化。EXISTS 适合相关子查询场景，JOIN 适合需要返回关联表字段的场景。实际执行中需通过 EXPLAIN ANALYZE 验证改写效果。

子查询上拉是指将子查询结果提前计算并缓存，适用于多次引用的场景。PostgreSQL 的 CTE（WITH 子句）默认物化，可显式声明 MATERIALIZED 或 NOT MATERIALIZED 控制物化行为。子查询下推则将过滤条件尽可能下沉到存储层，减少中间结果集大小。

分区表通过分区键裁剪可显著减少扫描数据量。例如按 created_at RANGE 分区的订单表，查询 WHERE created_at >= '2024-01-01' 可自动跳过历史分区。分区裁剪的前提是查询条件与分区键直接相关，且分区数量不宜过多，否则元数据管理开销会抵消裁剪收益。

5 执行计划解读与工具

MySQL 的 EXPLAIN 结果包含 id、select_type、table、partitions、type、possible_keys、key、key_len、ref、rows、filtered、Extra 等字段。type 字段的 const 表示通过主键或唯一索引单行访问，ref 表示通过非唯一索引等值访问，range 表示索引范围扫描，index 表示全索引扫描，ALL 表示全表扫描。Extra 字段的 Using filesort 表示需要额外排序，Using temporary 表示需要临时表存储中间结果。

回表操作在 Extra 字段表现为 Using index condition 或无 Using index 标记。随机 I/O 导致的性能下降可通过覆盖索引或延迟关联缓解。PostgreSQL 的 EXPLAIN (ANALYZE, BUFFERS) 输出中，实际执行时间和缓冲区命中率是判断索引有效性的关键指标。

慢查询日志记录执行时间超过阈值的语句，结合 performance_schema 的 events_statements_summary_by_digest 表可统计查询模式和资源消耗。PostgreSQL 的 pg_stat_statements 扩展提供类似功能，支持按调用次数、总耗时、平均耗时等多维度排序。

可视化工具可降低执行计划解读门槛。pt-visual-explain 将 MySQL 的 EXPLAIN 输出转换为树形结构，pgAdmin 的图形化执行计划支持节点展开和耗时占比展示。SQL Server Management Studio 的执行计划图形化界面可直接显示索引使用情况和并行度信息。

6 索引维护与演进

索引碎片源于页分裂和删除操作。MySQL 的 OPTIMIZE TABLE 可重建表和索引，PostgreSQL 的 REINDEX 可针对单个索引或整个数据库执行。重建操作需在低峰期执行，且大表重建可能引发长时间锁表，需评估业务影响。

冗余索引检测可通过 pt-index-usage 脚本分析慢查询日志，识别未被使用的索引。PostgreSQL 的 pg_stat_user_indexes 视图记录 idx_scan 字段，值为 0 的索引可能是冗余或废弃索引，需谨慎删除前确认无隐藏依赖。

在线 DDL 工具可避免长时间锁表。pt-online-schema-change 通过触发器和影子表实现无锁结构变更，gh-ost 采用 binlog 异步应用减少主库压力，pg_repack 通过重建表实现类似效果。这些工具均有一定风险，需在测试环境充分验证后再用于生产。

版本升级带来的新特性值得关注。MySQL 8.0 支持降序索引，可直接创建 (user_id, created_at DESC) 满足特定排序需求。PostgreSQL 13 优化了 B-Tree 索引的重复键处理，减少了索引膨胀。持续关注数据库版本更新日志，可提前规划索引策略演进。

7 真实案例拆解

电商订单表深分页场景中，原始查询 SELECT * FROM orders WHERE user_id = 123 ORDER BY created_at DESC LIMIT 100000, 10 执行时间超过 30 秒。通过建立 (user_id, created_at) 复合索引并改写为延迟关联，查询时间降至 50 毫秒。索引覆盖了 WHERE 和 ORDER BY 子句，避免了全表扫描和 filesort。

日志表按时间分区结合 BRIN 索引可降低存储成本。原始表每日新增 1000 万行数据，全表索引占用空间过大。改用按天 RANGE 分区，并在 created_at 列创建 BRIN 索引，索引大小仅为 B-Tree 的 1%，查询性能下降可接受范围内。BRIN 适合追加写入且按分区键查询的场景。

高并发账户表唯一索引冲突导致死锁的场景中，事务 A 执行 INSERT INTO accounts (id) VALUES (1)，事务 B 执行 INSERT INTO accounts (id) VALUES (2)，随后 A 执行 INSERT INTO accounts (id) VALUES (2)，B 执行 INSERT INTO accounts (id) VALUES (1)，形成死锁环。解决方案是统一插入顺序或使用 INSERT IGNORE/ON DUPLICATE KEY UPDATE 降低冲突概率。

PostgreSQL 函数索引加速 JSONB 查询的案例中，原始查询 SELECT * FROM events WHERE data->>'type' = 'login' 无法使用索引。通过创建 CREATE INDEX idx ON events ((data->>'type'))，查询可直接使用函数索引，执行时间从 2 秒降至 10 毫秒。函数索引的表达式需与查询条件完全一致，否则无法命中。

8 最佳实践 checklist

上线前索引 Review 需检查索引数量是否超过 5 个，单索引宽度是否超过 3 列，索引列选择性是否高于 0.1，查询是否可被覆盖索引满足。监控指标包括 Buffer Pool 命中率（目标 99% 以上）、索引命中率（目标 95% 以上）、锁等待时间（目标低于 1 秒）。

灰度发布策略要求先在从库验证索引效果，再在主库低峰期执行变更，并准备回滚脚本。文档管理需维护数据字典、ER 图和索引使用说明，确保团队成员理解索引设计意图和变更历史。

索引不是银弹，其收益最大化场景是读多写少且查询模式相对稳定的 OLTP 系统。持续观测和迭代优化是保持性能的关键，需建立常态化的慢查询 Review 机制和索引健康度监控体系。

推荐阅读《高性能 MySQL》、《PostgreSQL 修炼之道》以及官方文档中的优化器行为说明。实践过程中需结合具体业务场景和数据特征，灵活调整索引策略，而非生搬硬套理论规则。