# TiDB 可观测性解读系列：索引与算子执行性能优化实践

TiDB Club 2025年03月31日 17:00 北京



在分布式数据库的运维与优化中，可观测性是关键能力之一，它帮助数据库管理员快速定位问题、分析性能瓶颈并优化系统。TiDB 作为一款领先的分布式数据库，提供了丰富的可观测性工具，包括**索引观测**、**SQL 执行观测**、**数据热点观测**与**内存观测**等功能。

本文合集将深入探讨 TiDB 的索引优化和算子执行信息分析，帮助读者更好地理解和应用这些工具，提升数据库的性能和稳定性。

## 01

### 索引观测:快速识别无用索引与低效索引

索引设计对数据库性能优化至关重要。索引可以减少扫描的数据量，提高查询性能。然而，随着业务复杂性增加，索引设计可能存在问题，影响数据库效率：

**未使用的索引：**业务逻辑调整、数据变化或新索引创建，可能导致部分索引不再被优化器使用，成为“无用索引”。

**低效索引：**尽管索引被查询优化器选中，但扫描大量数据，导致 I/O 消耗高，优化效果不明显。

这些未使用或低效的索引若未及时清理，可能显著影响数据库性能和资源利用率。

## 索引优化的意义

**避免磁盘空间浪费：**索引占用磁盘空间，清理未使用索引可节省存储成本。

**降低 DML 操作的额外开销：**DML 操作需维护索引，移除低效索引可提升操作性能，尤其在高并发场景。

**提升查询性能：**优化低效索引能减少查询时的数据扫描量，提高查询效率。

**简化数据库维护：**清理不必要索引可简化数据库结构，提高维护效率。

## TiDB 对索引优化的支持

优化索引对数据库性能至关重要，但错误删除索引可能带来风险，导致 SQL 性能下降，甚至数据库性能崩溃。因此，删除索引的决策必须基于数据支持，通过有效手段验证，并在出现问题时能够快速回退。

TiDB 通过系统表 TIDB_INDEX_USAGE 和 schema_unused_index 的引入和不可见索引能力，帮助用户快速观测现有索引的状态，并实现删除索引前的验证。

### 系统表 TIDB_INDEX_USAGE

TIDB_INDEX_USAGE 从 v8.0.0 版本开始引入，记录了索引的关键运行指标，协助 DBA 制定优化策略。要识别无用索引，最直接的方法是查看索引被查询优化器选择的次数。**如果某个索引查询次数为零，说明它在当前 TiDB 实例中未被使用。**TIDB_INDEX_USAGE 提供了决策支持：

**QUERY_TOTAL：**记录访问某个索引的查询总次数。

**LAST_ACCESS_TIME：**记录该索引的最后访问时间。

**识别低效索引**相对复杂，主要通过观察索引的选择性来判断。TIDB_INDEX_USAGE 加入了以下字段来记录选择率分布情况：

**PERCENTAGE_ACCESS_：**因为实际选择率是离散的，在视图里依据选择率范围设计了不同的“桶(bucket)”，记录选择率落到该选择率范围的次数，以此判断索引的过滤效果。

ROWS_ACCESS_TOTAL：该索引扫描的总行数，用于衡量该索引对 I/O 的贡献。

### 系统表 schema_unused_indexes

为了方便用户直接查看结果，TiDB 还提供了一个 MySQL 兼容的视图 sys.schema_unused_indexes，**该视图列出了自所有 TiDB 节点启动以来，未被使用过的索引。**这张视图的数据来自 TIDB_INDEX_USAGE，请注意，由于 TIDB_INDEX_USAGE 在 TiDB 节点重启后会被清空，因此**在决策前需要确保节点的运行时间足够长。**

对于从旧版本升级到 TiDB v8.0.0 及更高版本的集群，sys schema 以及包含的视图需要手动创建，请参考官方文档进行操作。

### 不可见索引 (invisable indexes)

清理索引存在一定的风险，一旦错误地删除索引，重建索引和统计信息收集可能要花费很长时间，为降低风险，推荐先将索引设置为“不可见”状态。**设置为不可见后，优化器将不再使用该索引，但索引的统计信息仍会被更新维护，可快速恢复为“可见”。**DBA 可以先将要删除的索引设置为不可见，观察一段时间后再决定是否真正物理删除。

## 索引优化实战

根据观测到的索引所处的状态，我们可以灵活选择索删除或优化相关索引。在这个部分将介绍如何安全、有效地识别优化 TiDB 问题索引。

### 删除未使用的索引

**步骤 1：检查 TiDB 节点的运行时间**

**步骤 2：获取 schema_unused_indexes 的输出**

**步骤 3：将索引设置为不可见**

**步骤 4：观察数据库表现**

**步骤 5：恢复索引可见（如果出现性能回退）**

**步骤 6：安全删除未使用的索引**

## 识别并优化低效索引

一些效率不高的索引不仅无法提升查询速度，还可能引发额外的 I/O 操作，增加查询的响应时间。利用 TIDB_INDEX_USAGE 表中汇总的数据，数据库管理员（DBA）能够根据实际需求，识别出低效索引后，DBA 需要评估并进行优化，这里分为几种情况：

**执行计划选择问题：**如果数据库中存在更优的索引却没法被优化器选到，可以**从数据库层面对优化过程进行调优。**

**设计建模问题：**如果不是执行计划选择的问题，则需要**从设计角度进行优化。**常见的优化思路有**调整索引设计、调整查询设计、调整应用建模等。**

## 总结

DBA 在执行索引优化时应遵循以下最佳实践：

定期检查索引使用情况，尤其是对于大规模数据库。

确保用于决策的统计数据涵盖足够长的业务周期，避免误判。

通过设置索引为“不可见”来验证是否会影响查询性能，降低删除索引可能造成的风险。

创建和删除索引应选择在业务低峰时段进行。

通过遵循这些最佳实践，TiDB 用户不仅能保持系统的稳定性，还能实现高效的索引管理和优化，确保数据库的高效运行。

点击此处 | 查看原文

# 02

## 算子执行信息性能诊断案例分享

### 算子执行信息介绍

通常我们可以用 explain analyze 语句获得算子执行信息。explain analyze 会实际执行对应的 SQL 语句，同时记录其运行时信息，和执行计划一并返回出来，记录的信息如下图所示。

| 属性名 | 含义 |
| --- | --- |
| actRows | 算子实际输出的数据条数。 |
| execution info | 算子的实际执行信息。time 表示从进入算子到离开算子的全部 wall time，包括所有子算子操作的全部执行时间。如果该算子被父算子多次调用 （loops），这个时间就是累积的时间。loops 是当前算子被父算子调用的次数。 |
| memory | 算子占用内存空间的大小。 |
| disk | 算子占用磁盘空间的大小。 |

不同算子的 execution info 可以通过 TiDB 文档 了解。有时候一些 SQL 的性能问题是偶发的，这会增加我们直接使用 explain analyze 来分析的难度。通常，我们可以通过 TiDB Dashboard 的慢日志查询 页面，快速定位并查询到问题 SQL 的详细执行信息。

### 案例和问题

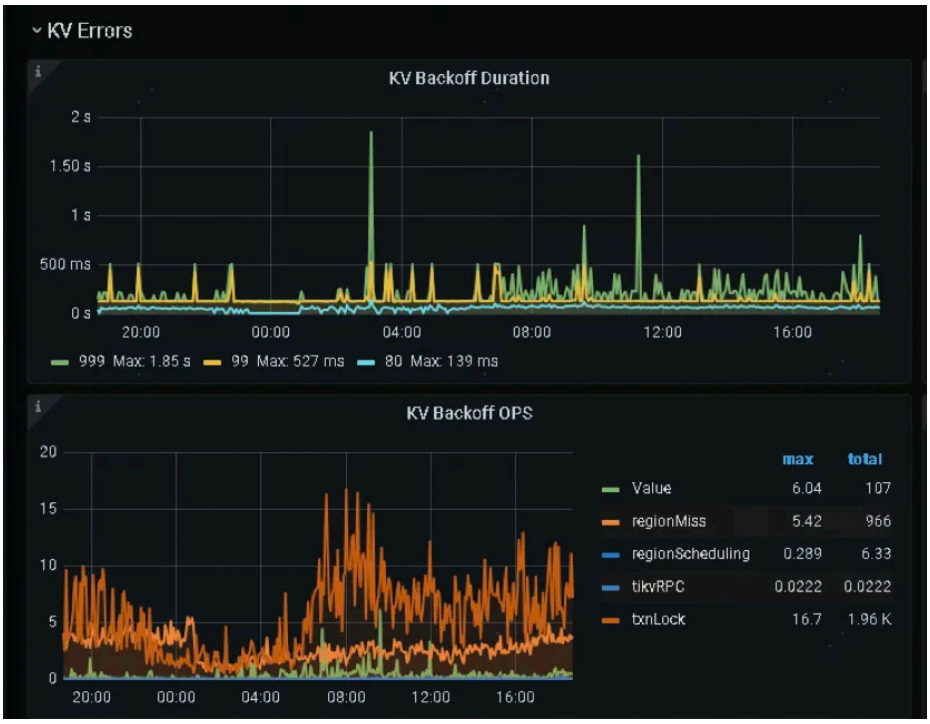
接下来，将通过一些具体案例来探讨诊断问题的思路和方法。

#### 查询延迟抖动

**实际案例：**以一个客户遇到过的点查询性能抖动问题为例，点查询的延迟偶尔会超过 2s。

**诊断分析：**在 txnNotFound_backoff 项中，它记录了因残留事务触发的重试信息。这里显示总共进行了 12次重试，累计耗时 2.51s，与 ResolveLock 项的 2.52s 基本一致。因此，我们可以初步推测：点查询可能读取到了残留事务的锁，尝试 resolve lock 时发现锁已过期，进而触发了锁清理操作，这一过程导致了较高的查询延迟

接下来，我们可以通过监控数据进一步验证这一推测：




## 算子并发度

**实际问题：**系统中设置了 tidb_executor_concurrency 为 5 以控制算子的并发度；

同时设置了 tidb_distsql_scan_concurrency 为 15，用于限制每个读取算子的最大线程数。那么，cop_task 和 tikv_task 是如何与这两个参数相对应的呢？

**问题分析：**在 TiDB 中，cop_task 和 tikv_task 是不同维度的执行信息。tikv_task 描述单个 TiKV 算子的执行情况，而 cop_task 描述整个 RPC 任务，包含多个 tikv_task。

例如，IndexLookUp 算子的 cop_task 并发度由 tidb_distsql_scan_concurrency 决定，table_task 的并发度由 tidb_executor_concurrency 决定。

## max 换成 min，慢了好几十倍

**实际案例：**一条对索引列求 max 值的 SQL 花费 100 毫秒左右，换成求 min 值，却需要花费 8s 多时间。

**诊断分析：**一般先从上到下看每个算子本身（即去除掉等待子算子数据的时间后）的执行时间，再去寻找对整个查询性能影响最大的算子。

不同类型单个算子的算子执行时间计算方法有**类型为 "root" 的算子、类型为 "root" 的算子、类型为 "mpp[tiflash]"、"cop[tiflash]" 或 "batchcop[tiflash]" 的算子**等。

对于包含多个子算子的复合算子，其 **execution info** 中通常会详细列出各个子算子的执行信息。通过分析这些子算子的执行信息，我们可以精准判断哪个子算子对整体性能的影响更大，从而更有针对性地进行优化。

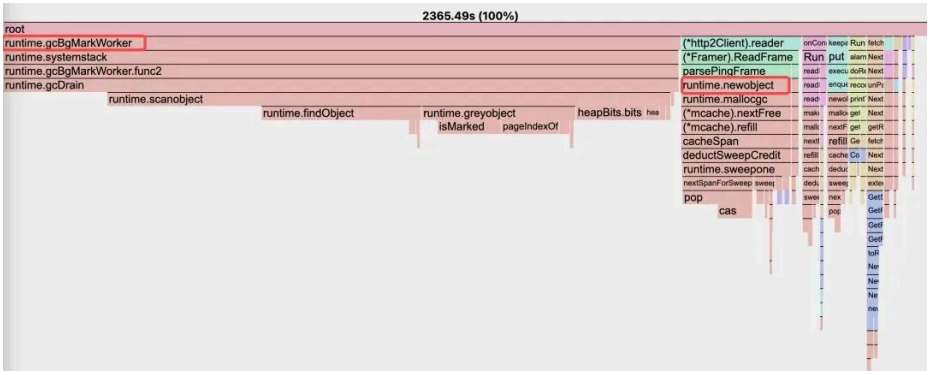
案例中在“min”场景中，最小的 keys 附近恰好存在大量已被删除但尚未回收的 keys。因此，系统在读取过程中不得不扫描大量无用数据，直到最终找到第一条有效数据，这导致了显著的性能开销。

## 换个日期，查询慢了 160 倍还没跑出来

**实际案例：**查询 12 月 20 日数据的 SQL 语句仅耗时 25 分钟便完成，而查询 12 月 21 日数据的 SQL 语句却运行了超过 40 小时仍未完成，最终只能手动终止。

**诊断分析：**从上至下逐一分析，找出两次查询中主要执行时间差异的来源。可以看到，Insert_1、Projection_7 和 HashAgg_16 的执行时间均在秒级，因此可以排除这些算子的嫌疑。进一步查看 table_task 的执行细节，基本可以确定问题出在 table_task 上，即对应的 HashAgg_9算子。

HashAgg_9 的总执行时间并非主要消耗在 **cop task** 的执行上，而更可能是花费在从 **cop task response** 中读取数据的过程中。结合 CPU profiling 的结果，可以看到 Go runtime 的 GC（垃圾回收）占用了大约 **80%** 的 CPU 时间。




**进一步观察 heap profiling 的结果，可以最终确认问题根源是 issue44047**。该问题已经在 TiDB 6.5 版本中修复。

## 未来展望

在 TiDB 9.0 版本中，我们将进一步丰富算子执行信息，提升系统的可观测性，具体改进包括：

**算子执行时间的细化：**在 9.0 版本中，time 被修正为从进入算子到离开算子的完整 wall time，包括所有子算子的执行时间。

**并发执行时间的区分：**在 9.0 版本中，这些累加的时间信息（如 time、open 和 close）将被替换为 total_time、total_open 和 total_close，以更清晰地反映并发执行的真实耗时。

**TiFlash 执行中的等待时间信息补充 ：**在 TiFlash 的执行信息中，新增了 minTSO_wait、pipeline_breaker_wait、pipeline_queue_wait 等待时间的统计。

通过这些改进，TiDB 9.0 版本将为用户提供更全面、更准确的执行信息，帮助更好地诊断和优化查询性能。

点击此处 | 查看原文