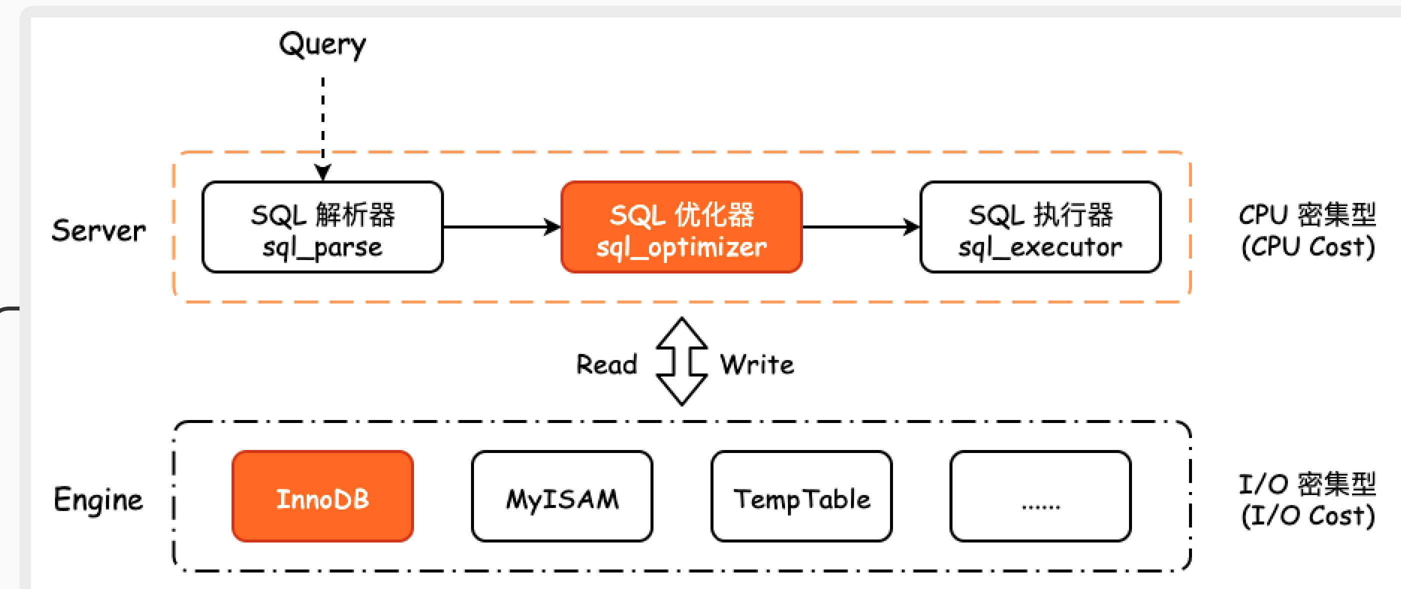
Cost-Based Optimizer

MySQL 查询流程



- 忽略缓存
- MySQL 在执行一条 Query SQL 时，由 Server 层和 Engine 层共同完成
  - Server 层
    - Server 层 包括 SQL 解析器、SQL 优化器以及 SQL 执行器，用于负责 SQL 语句的具体执行流程，以及对结果集进行过滤、排序以及键值的比较等
    - 对于 Server 层而言，更多的是对内存中的数据进行比较、筛选以及排序等，因此是一个 CPU 密集型的任务
  - Engine 层
    - Engine 层主负责存储具体的数据，例如 InnoDB、MyISAM 等存储引擎，以及在内存中存储临时结果集的 TempTable 等
    - 对于 Engine 层而言，绝大多数时刻都在和硬盘打交道，将索引树上的数据发送给 Server 层，因此是一个 I/O 密集型的任务
- 在 Server 层中，SQL 优化器是我们需要重点关注的内容，因为一条 SQL 到底该怎么执行、选择哪一列索引、怎么进行子查询，都是由 SQL 优化器决定的

基于成本的优化器

- 优化器在做出选择时，比如使用哪一个索引、是全表扫描还是走索引，都是基于成本来进行的。所谓的成本，包括从存储引擎取出数据的成本，创建临时表的成本，索引键值比较的成本，排序的成本等等
- 在 MySQL 查询流程中我们有描述过一条 SQL 的执行分为 Server 层和 Engine 层，那么这两层会有分别的成本计算
- Server

```
mysql> select * from mysql.server_cost;
+------------------------------+------------+---------------------+---------+---------------+
| cost_name                    | cost_value | last_update         | comment | default_value |
+------------------------------+------------+---------------------+---------+---------------+
| disk_temptable_create_cost   |       NULL | 2020-12-14 10:32:36 | NULL    |            20 |
| disk_temptable_row_cost      |       NULL | 2020-12-14 10:32:36 | NULL    |           0.5 |
| key_compare_cost             |       NULL | 2020-12-14 10:32:36 | NULL    |          0.05 |
| memory_temptable_create_cost |       NULL | 2020-12-14 10:32:36 | NULL    |             1 |
| memory_temptable_row_cost    |       NULL | 2020-12-14 10:32:36 | NULL    |           0.1 |
| row_evaluate_cost            |       NULL | 2020-12-14 10:32:36 | NULL    |           0.1 |
+------------------------------+------------+---------------------+---------+---------------+
```

  - 创建磁盘临时表的成本
  - 磁盘临时表中每条记录的成本
  - 索引键值比较的成本
  - 创建内存临时表的成本
  - 内存临时表中每条记录的成本
  - 记录间的比较成本
  - MySQL 认为创建临时表的开销最大，因为当我们进行 SQL 查询时，需要尽可能地避免使用临时表。同时，虽然索引键值比较的成本和记录间比较的成本非常低，但是我们查询时可能会涉及到多条数据，那么此时比较成本将以线性进行增长
- Engine

```
mysql> select * from mysql.engine_cost;
+-------------+-------------+------------------------+------------+---------------------+---------+---------------+
| engine_name | device_type | cost_name              | cost_value | last_update         | comment | default_value |
+-------------+-------------+------------------------+------------+---------------------+---------+---------------+
| default     |           0 | io_block_read_cost     |       NULL | 2020-12-14 10:32:36 | NULL    |             1 |
| default     |           0 | memory_block_read_cost |       NULL | 2020-12-14 10:32:36 | NULL    |          0.25 |
+-------------+-------------+------------------------+------------+---------------------+---------+---------------+
```

  - 从硬盘读取一页的成本
  - 从内存读取一页的成本
  - 也就是说，MySQL 认为读取硬盘的速度要比读取内存的速度慢 4 倍。4 倍的差异对于 SSD 来说是正常的，但是对于 HDD 来说，这个值就需要人为地调整到更大的值
- 😀 PS: 以下测试均在 MySQL 8.0.23 上进行，存储引擎为 InnoDB
- 一个简单的例子

primary key　index=True　unique=True

id	name	user_id
1	Auz	100001
2	Buz	100002
3	Cuz	100003
4	Duz	100004

```
mysql> explain format=JSON select * from user\G;
*************************** 1. row ***************************
EXPLAIN: {
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1.40"                # 总花费成本
    },
    "table": {
      "table_name": "user",
      "access_type": "ALL",               # 使用全表扫描的方式完成
      "rows_examined_per_scan": 4,        # 总扫描行数
      "rows_produced_per_join": 4,        # 满足条件的预估行数
      "filtered": "100.00",               # 预估百分比数，预估行数/总扫描行数 * 100%
      "cost_info": {
        "read_cost": "1.00",              # Engine Cost (I/O Cost)
        "eval_cost": "0.40",              # Server Cost (CPU Cost)
        "prefix_cost": "1.40",            # read_cost + eval_cost，即总成本
        "data_read_per_join": "384"       # 读取记录的总字节数
      },
    }
```

  - read_cost
    - 由于 user 表中没有数据在缓存中，所以必须从硬盘中将对应的 page 读取至内存中，因此成本为 1，而不是 0.25
    - 由于 user 表中数据较少，所以内存可容纳全表的数据，那么当我们执行 select * from user 之后，再执行 EXPLAIN 时将会发现其 read_cost 降为 0.25
  - eval_cost — 当我们取出一页的数据以后，MySQL 还需要在 Server 层中进行扫描提取，因为有 4 条记录，所以成本为 4 * 0.1 = 0.4

为什么 MySQL 会"选错"索引

- 在使用 MySQL 的时候，我们可能会碰到这样的例子: 明明对应的字段上存在索引，但是 MySQL 依然使用全表扫描的方式来执行该 SQL 语句，是不是优化器有 Bug?
- MySQL 不使用索引的原因只有一个，那么就是 MySQL 认为不使用索引的成本相较于使用索引的成本更低，主要原因就在于回表
- ❶ 未使用建立的索引

约 3W 行

id	created_at	updated_at	user_id	order_id	status
1	2019-12-23 10:38:32	2019-12-23 10:39:40	100001	201912132145963	50
......	......	......	......	......	......

  - 可以认为这些字段上全部被添加了索引
  - created_at 范围在 2019-05-01 12:00:00 ~ 2021-06-21 12:00:00
  - 假设我们执行 explain format=JSON select * from orders where created_at between '2019-06-01' and '2021-06-01'; 的话，会发现 MySQL 使用了全表扫描的方式执行了该 SQL 语句，尽管 created_at 字段上存在索引
  - 原因在于 MySQL 认为 '2019-06-01'~ '2021-06-01' 这个时间范围包括了表中绝大部分数据，如果使用 created_at 这一二级索引的话，那么在索引树上查询以后，还需要进行回表查询，假设这种方式的成本为 X。而后优化器发现使用全表扫描的方式所需成本为 Y，并且 Y < X，因此会选择直接扫描聚簇索引然后再进行过滤
  - 更进一步地，如果我们将查询范围缩小的话，就会发现，MySQL 使用了 created_at 上的二级索引
- ❷ 有限状态上的索引
  - 还是上面的例子，如果我们对 status 字段添加索引，并且仅使用 status = 50 进行查询。当数据量较大时，MySQL 仍然会使用全表扫描的方式。原因在于 status 字段中的状态是有限的
  - 假如 status 的枚举值包括 10、20、30、40、50 这 5 个的话，那么 MySQL 就会认为对应状态的数据是均匀分布的，也就是状态为 10 的数据占有 1/5，状态为 50 的数据同样占用 1/5。但是，如果 50 表示异常订单的话，那么其占比可能是非常之小的，也就是 select * from orders where status = 50; 使用索引可能会更快
  - 此时，我们就需要主动地告知 MySQL 我们的数据分布，让优化器更好的进行工作。很简单，只需要在 status 上建立一个直方图即可
    - ANALYZE TABLE orders UPDATE HISTOGRAM ON status;
    - 直方图为 8.0 的新特性，5.7 版本并不支持
- 一般来说我们只会对高选择度的字段添加索引，诸如性别、状态这样的字段不添加索引。但是，如果我们一定要根据低选择性的字段进行查询，并且存在数据倾斜时，可通过建立索引+直方图的方式帮助优化器进一步地校准执行计划