"慢SQL"治理的几点思考

原创 邱腾龙 转转技术 2025年03月26日 19:50 广东

- 一.背景
- 二.MySQL是如何评估成本的？
- 三.即使加了索引，也没有起作用
- 四.内存碎片也是一个值得关注的问题
- 五.前缀索引的坑
- 六.索引合并
- 七.有时候SQL没啥问题，但还是报了慢查询？
- 八.总结

一.背景

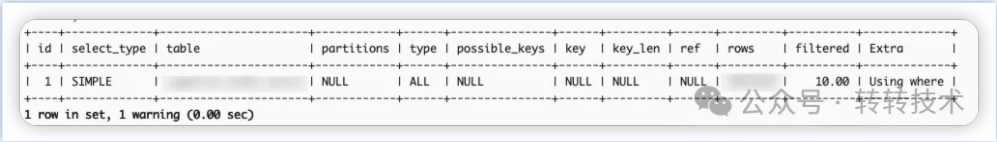
今年初团队开始推行“服务稳定性问题治理专项”。通过错误日志、慢SQL、接口性能等各项指标的优化，进一步提升系统稳定性与可靠性。在此契机之下，本文将从“慢SQL治理”的角度，通过部分实际案例，分析其原理，做一些阶段性总结和思考。

二.MySQL是如何评估成本的？

某日线上突发告警，发现有一条慢SQL

```
select * from xxxx_supplier_extra
where column_key = 'xxxx'
```

作为有“临床经验”的老司机，第一直觉是先用explain分析。explain用了毫秒级的时间就输出了这样一份执行计划，果然它很快，两分钟都不到。



```
+----+-------------+-------------------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table             | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+-------------------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | [illegible]       | NULL       | ALL  | NULL          | NULL | NULL    | NULL | [illegible] |    10.00 | Using where |
+----+-------------+-------------------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```


通过“key=null”，我们不难发现，这条SQL走了全表扫描。这是一条简单的SQL案例，我们借用来分析MYSQL背后的成本评估原理。

执行SQL前，优化器会先选择它认为成本最低的方案。正如同有电梯坐电梯，没电梯爬楼梯。

由于没有可供选择的索引，执行器选了全表扫描。查询成本一般由IO成本和CPU成本组成。众所周知，表数据是存储在磁盘的，每次读磁盘上的数据都 产生IO，受制于磁盘的物理属性，IO往往占查询成本的大头。举个例子，key_name="input_time"这行记录存储在磁盘页A中，读完这行后，又读了页B中的数据key_name="input_state"。此时多了一次IO，成本就有多一些。

因为全表扫描是发生在聚簇索引上，首先会估算整个聚簇索引占用的页面数，以及表的记录数，再计算IO成本和扫描成本（可以理解为CPU成本）。

计算口径如下：

```
IO成本：页面数 x io_block_read_cost（IO成本数 默认0.25） + 1.0
CPU成本：记录数 x cpu_tuple_cost （扫描成本数 默认0.1）
总成本 = IO成本 + CPU成本
```

MySQL的IO成本默认基于随机IO计算（`io_block_read_cost=1.0`），而非顺序IO。这里因为全表扫描是顺序IO，io_block_read_cost的默认值则为0.25。表记录越多，占用的页数则随着增长，其查询成本也就不断累加。

三.即使加了索引，也没有起作用

看到慢 SQL 直接加索引就好吗？基于前面全表扫描的原理，是否看到慢 SQL 直接加索引就完事了？

以前面SQL为例，当我们为‘column_key’字段加索引后，测试环境 explain 分析能命中索引，但上线后还是咔咔咔出现慢查询。

这是因为如果索引字段的区分度不够，优化器会认为查找成本过大，此时还是选择走全表扫描。而测试环境表记录较少的情况下，优化器觉得回表开销不大，就能命中索引，这也解释了为什么两者的执行计划不同。

索引能否命中往往与查询条件以及数据分布有关。

如何在索引设计之初就规避此类问题？

网上一些文章会提到基于该列的业务属性来区分，例如性别字段只有两个值：“sex=男，sex=女”。由于大部分记录都拥有相同值，数据区分度不大，所以容易成为低效索引。

除此之外，可以使用该语句计算区分度：

```
SELECT COUNT(DISTINCT column_name )
/ COUNT(*)
```

区分度低于10%的字段避免单独建索引。对于联合索引而言，也应尽量将区分度高的字段放在前面。

值得注意的是，即使该字段的区分度能够建立索引。也要根据已有索引和查询场景做综合取舍，要避免在同一个表上堆砌过多索引。

四.内存碎片也是一个值得关注的问题

内存碎片对优化器的影响

上文提到了成本估算是基于页数以及记录数计算的，这些数据来源于库中的统计信息。当内存碎片过大时，如果出现库表的统计信息未及时更新，也会因为优化器评估的结果与实际差距太大，从而影响实际执行效果。

内存碎片导致慢查询

举个例子：某日xxxx_price表产生慢查询告警，该表作为统计数据表，其查询SQL较简单，单纯从SQL上分析并没有太多问题。

联想到前阵子，该表由于历史原因，积压了2亿+无效数据。后面做了批量删除清理，由此推断可能是内存碎片导致的。

通过查看表的TABLE STATUS

```
SHOW TABLE STATUS LIKE 'xxx_price'
```

输出结果显示：Data_free=54835281920。碎片占了大量空间。

当内存碎片过多时，首当其冲会让物理IO被放大。原本“id=1，id=10，id=15”这三条记录读一次数据页就能够拿到，现在由于这几条数据被分散在多个数据页中，从而引发IO次数增多。同时，数据页是加载到缓冲池（Buffer Pool）里面的，这也会导致缓存命中率下降。

一般情况下，有一定的内存碎片是正常情况。但当内存碎片的占比过高时，则需要关注。

为什么频繁删除会出现内存碎片问题？

五.前缀索引的坑

某日发现线上出现一些重复异常，显示查询某参考价表的数据重复了，通过排查insert的两条数据，发现其实并没有重复。看了表结构才发现，原来表某个字段加了唯一索引，且唯一索引键使用了前缀索引。

```
unique (cate_id, brand_id,
model_id, key_props(10))
```

由于 key_props字段使用了前缀索引，因此索引树的叶子节点，并没有完整地存储整个字符串，而是截取字符串前面N个字符。这可以有效地节省索引空间。但这里的问题是使用了唯一索引，导致两个不同的字符串，只是前缀相同就触发重复冲突。

一般对于长度过长的字段，加前缀索引是一种选择，但像案例中在唯一约束中使用前缀索引，则需要保证前缀唯一性

六.索引合并

除了在单个索引下检索数据，其实还有可能在多个索引上检索。在符合特定条件下，通过索引合并，能够减少回表带来的消耗。如：

```
select * from xxx_supplier_order
where k1 = '123' and k2 = '345'
```

假设xxx_supplier_order的主键是id字段。k1和k2分别是两个独立的索引字段。当满足一定条件时（k1的叶子结点上id是有序的，k2也是id有序）。此时MYSQL可以根据 k1 = '123'在索引上检索出id，再根据 k2 = '345'在索引上检索id。并将两次检索的id取交集，就可以筛选出符合条件的id并回表执行。

这样做的好处在于

1.要同时满足 k1 = '123' and k2 = '345'的记录，其id必然存在于两个索引树上，通过交集，筛选出少量符合条件的id才去回表，理论上能够有效减少回表的次数。

2.id有序性有利于取交集操作，如某次检索。从k1上读到id=1，再从k2读到id=2，此时就可以判定id=1不满足k2的条件。另外，通过有序id，也能够确保每次回表能够有序，避免随机IO。

七.有时候SQL没啥问题，但还是报了慢查询？

如果SQL没有问题，那么关注点可以放在mysql实例的资源开销上了。因为造成慢查询的原因不单只是SQL本身，有可能是磁盘负载，CPU以及网络 等方面的资源不足引起了。举个例子：

某统计服务数据库，其库表大部分数据源自大数据平台的异步交换任务。某个时间段有多个交换任务往库表里面导入大批量数据，从而引发了磁盘等资源的负载增加，带来慢查询。

另外有时候事务的问题也需要关注。比如当长事务导致Undo Log膨胀时，容易使得扫描效率降低。同时Buffer Pool中缓存页因旧版本数据过多，其缓存命中率也会下降。我们可以通过`SHOW ENGINE INNODB STATUS`中`History list length`值是否飙升，加以判断。

八.总结

本文试图通过一些案例，分析其背后的原理。至于覆盖索引，联合索引等其它内容，相信网上有很多类似的内容，这里不多赘述。慢SQL治理是一个值得关注的问题。重要的是理解MySQL索引，事务等方面执行原理，然后现实使用场景中灵活分析和运用。

关于作者

邱腾龙 B2C供应链研发工程师

#索引 1 #慢SQL 1 mysql 9 性能优化 4 Java 4