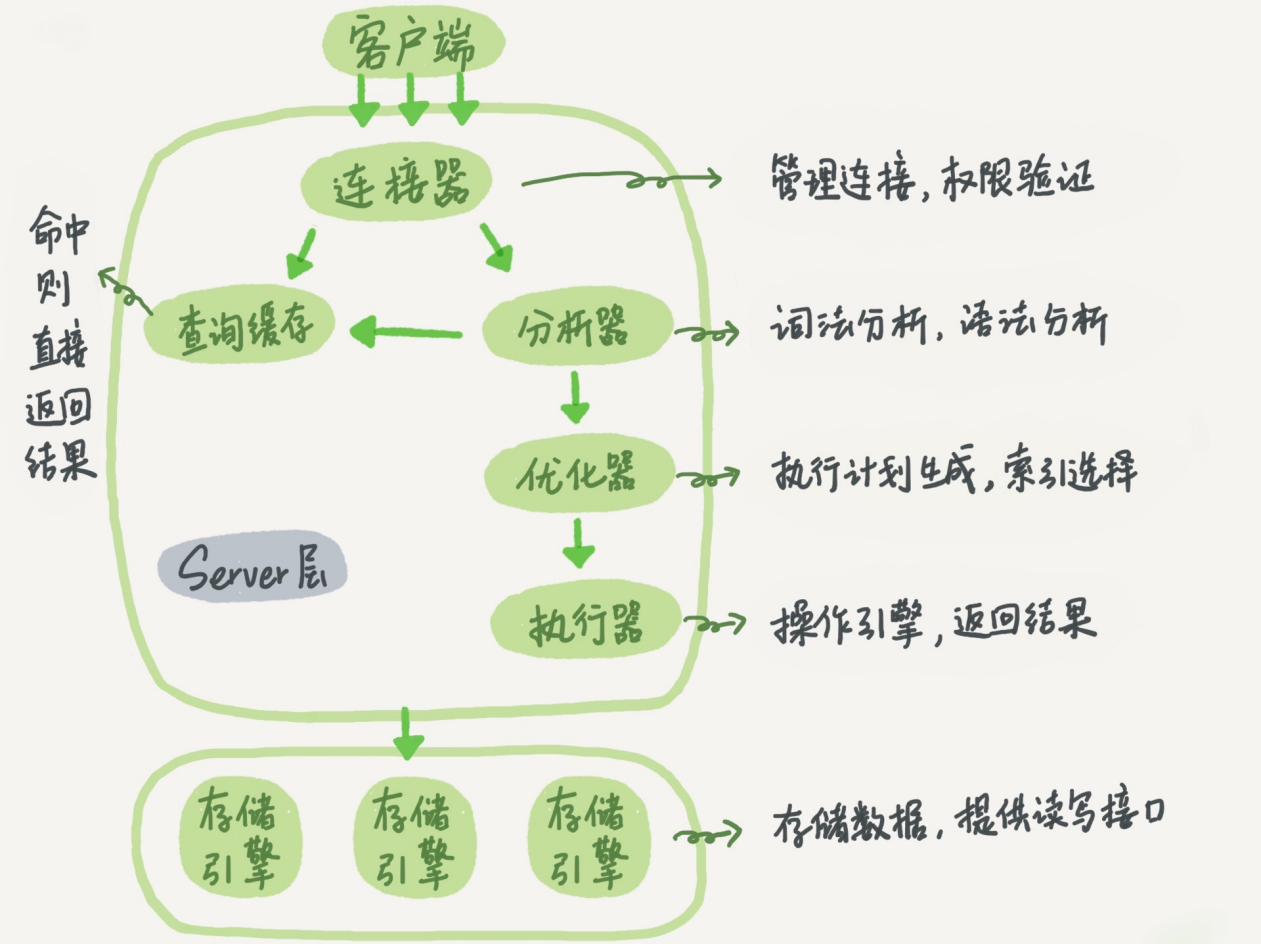
MySQL 的性能: 查询优化

> **Organization**: 千锋教育 Python 教学部
> **Date** : 2019-10-22
> **Author**: 张旭

查询操作在实际开发中用的最多，程序调用的也最多。对于线上的程序，性能的主要压力便来自于查询，尤其是复杂查询。



常用的查询优化策略有：减少数据访问、重写SQL、重新设计表、添加索引4种。

一、减少数据访问

减少数据访问一般考虑的方向是：

1. 减少应用程序对数据库的访问:

   数据库和应用程序之间是网络通信，每次通信都有 I/O, 所以应该减少通信次数。
   能不通信的尽量不通信，能一次完成的尽量不要分多次。
   例如: 为应用程序增加缓存

2. 减少实际扫描的记录数:

   查询时扫描的行越多，查询越慢，尽量加以控制
   例如: 尽量配合条件去使用 limit、offset, 对比一下这两句的性能差异:

```
select * from xxx limit 10 offset 10000;       -- 慢
select * from xxx where id > 10000 limit 10;   -- 快
```

3. 减少数据的获取量:

   例如: 只需要获取几列数据时，不要获取全部列，避免使用

```
select * from ...
```

二、重写 SQL 语句

由于复杂查询严重降低了并发性，因此为了让程序更适于扩展，我们可以把复杂的查询分解为多个简单的查询。一般来说多个简单查询的总成本是小于一个复杂查询的。

对于需要进行大量数据的操作，可以分批执行，以减少对生产系统产生的影响，从而缓解复制超时。

由于MySQL连接 (JOIN) 严重降低了并发性，对于高并发高性能的服务，应该尽量避免连接太多表，如果可能，对于一些严重影响性能的SQL，建议程序在应用层就实现部分连接的功能。

这样的好处是: 可以更方便、更高效地缓存数据，方便迁移表到另外的机器，扩展性也更好。

连接语句的问题

由于连接的成本比较高，因此对于高并发的应用，应该尽量减少有连接的查询，连接的表的个数不能太多，连接的表建议控制在4个以内。

互联网应用比较常见的一种情况是，在数据量比较小的时候，连接的开销不大，这个时候一般不会有性能问题，但当数据量变大之后，连接的低效率问题就暴露出来了，成为整个系统的瓶颈所在。

所以对于数据库应用的设计，最好在早期就确定未来可能会影响性能的一些查询，进行反范式设计减少连接的表，或者考虑在应用层进行连接。

连接语句的优化

1. ON、USING子句中的列确认有索引。如果优化器选择了连接的顺序为B、A，那么我们只需要在A表的列上创建索引即可。

   例如，对于这个查询语句:

   ```
   1 SELECT B.*,A.*FROM B JOIN A ON B.col1=A.col2;
   ```

   MySQL会全表扫描 B 表，对 B 表的每一行记录探测 A 表的记录(利用 A 表 col2 列上的索引)。

2. 最好是能转化为 `INNER JOIN`，`LEFT JOIN` 的成本比 `INNER JOIN` 高很多。

3. 使用 `EXPLAIN` 检查连接，留意 `EXPLAIN` 输出的 rows 列，如果 rows 太高，比如几千、上万，那么就需要考虑是否索引不佳或连接表的顺序不当。

4. 反范式设计，这样可以减少连接表的个数，加快存取数据的速度。

5. 考虑在应用层实现连接。

   对于一些复杂的连接查询，更值得推荐的做法是将它**分解**为几个简单的查询，可以先执行查询以获得一个较小的结果集，然后再遍历此结果集，最后根据一定的条件去获取完整的数据，这样做往往是更高效的。

   因为我们把数据分离了，更不容易发生变化，更方便缓存数据，数据也可以按照设计的需要从缓存或数据库中进行获取。

例如，对于如下的查询:

```
SELECT a.* FROM a WHERE a.id B;
```

如果id=1~15的记录已经被存储在缓存(如Memcached)中了，那么我们只需要到数据库查询:

```
SELECT a.* FROM a WHERE a.id=16;
```

和

```
SELECT a.* FROM a WHERE a.id=17;
```

而且，把IN列表分解为等值查找，往往可以提高性能。

6. 一些应用可能需要访问不同的数据库实例，这种情况下，在应用层实现连接将是更好的选择。

Group、Order、Distinct 的优化

`GROUP BY`、`DISTINCT`、`ORDER BY` 这几类子句比较类似，`GROUP BY` 默认会进行 `ORDER BY` 排序

优化方向如下:

1. 尽量对较少的行进行排序
2. 如果连接了多张表，`ORDER BY` 的列应该属于连接顺序的第一张表。
3. 利用索引排序。
4. GROUP BY、ORDER BY语句参考的列应该尽量在一个表中，如果不在同一个表中，那么可以考虑冗余一些列，或者合并表。
5. 需要保证索引列和ORDER BY的列相同，且各列均按相同的方向进行排序。

优化子查询

对于数据库来说，在绝大部分情况下，**连接查询会比子查询更快**。

使用连接的方式，MySQL优化器一般可以生成更佳的执行计划，可以预先装载数据，更高效地处理查询。

而子查询往往需要运行重复的查询，子查询生成的临时表上也没有索引，因此效率会更低。

一些商业数据库可以智能地识别子查询，转化子查询为连接查询。但MySQL对于子查询的优化一直不佳，就目前的研发实践来说，子查询应尽量改写成JOIN 的写法。

优化 limit 子句

limit 子句的问题主要出在 offset 上, offset的值很大，效率就会很差。

一般可以进行如下调整：

1. 从应用程序上进行调整，限制页数，只显示前几页，超过了一定的页数后显示 “获取更多”
2. 能避免 offset，尽量避免
3. 不能避免的时候使用条件约束查询结果的范围，不要匹配太多。

优化IN列表

- IN 列表不宜过长，最好不要超过200。
- 对于高并发的业务，小于几十为佳。

三、重新设计表结构

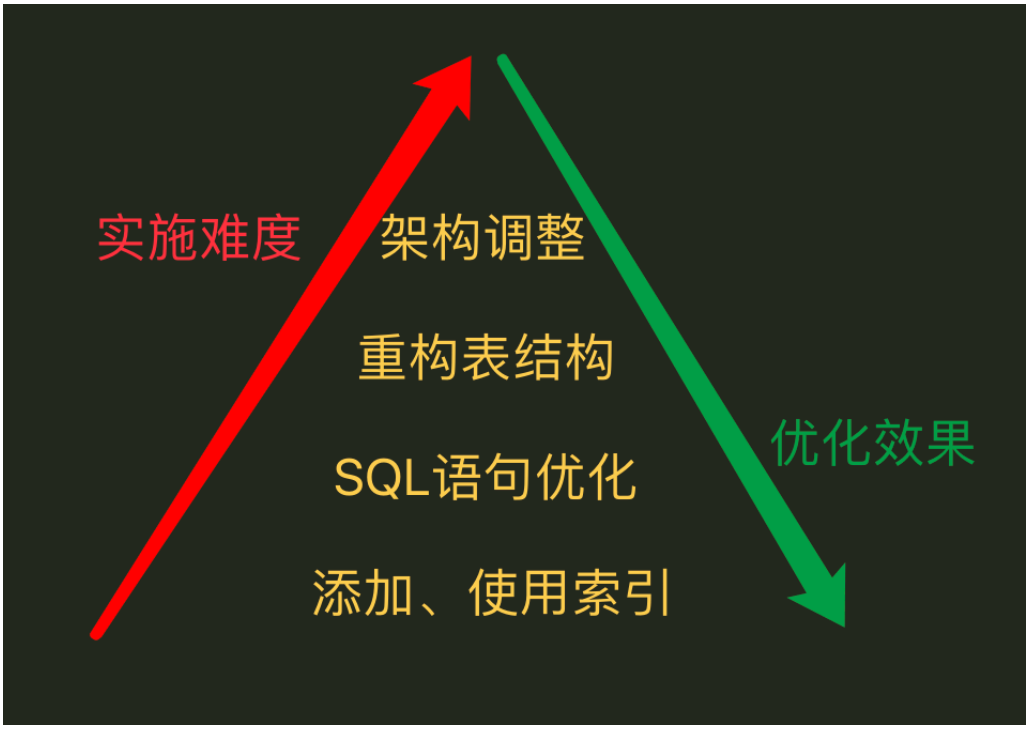
有些情况下，我们即使是重写SQL或添加索引也是解决不了问题的，这个时候可能要考虑更改表结构的设计。

比如，可以增加一个缓存表，暂存统计数据，或者可以增加冗余列，以减少连接。

优化的主要方向是进行反范式设计。

四、添加索引

生产环境中的性能问题，可能 80% 的都是索引的问题，所以优化好索引，就已经有了一个好的开始。



好的索引能起到事半功倍的效果，但是在实际操作时也要注意一些事情，不合理的 SQL 语句会导致索引失效。

1. 在where子句中进行null值判断的话会导致引擎放弃索引而产生全表扫描

   ```
   SELECT id FROM table WHERE num is not null;
   ```

   在建立数据库的时候因尽量为字段设置默认值, 如int类型可以使用0, varchar类型使用 ''
   当你在指定类型大小如int(11)时, 其实空间就已经固定了, 即时存的是null 也是这个大小

2. 避免在 where 子句中使用 `!=`, `<>` 这样的符号, 否则会导致引擎放弃索引而产生全表扫描

   ```
   SELECT id FROM table WHERE num != 0;
   ```

3. 避免在where子句中 `=` 的左边使用表达式操作或者函数操作

```sql
SELECT id FROM table WHERE num / 2 = 1;
-- 应换成
SELECT id FROM table WHERE num = 2;
-- 函数操作也是同理
SELECT id FROM table WHERE SUBSTRING(name, 1, 2) =
'wise'; -- 这种尽量避免
```

4. 避免在 where 子句中使用 like 模糊查询

```sql
-- 放弃索引，全表扫描
select count(1) from `user` where nickname like
'%ab';

-- 使用索引
select count(1) from `user` where nickname like
'ab%';
```