# SQL 调优最佳实践 「值得收藏」

### 写作背景

网上一直在说 SQL 调优，到底什么是 SQL 调优？是不是觉得自己知道但又很模糊说不清楚，面试被问没有真实的案例，并不具备说服力。本文以为详细案例给大家解读。

开始之前，先回答什么是 SQL 调优，SQL 调优是为了让 SQL 获得更好的性能、查询更快、资源消耗少。简单概述就是别人写的 SQL 查询耗时 10s，扫描数据 100w 行；你写的 SQL 耗时 100ms，扫描 1w 行，相比较你的 SQL 更优。

### SQL 优化基本原则

为了让 SQL 获得更好的性能，应该遵循下面几个原则

1. 扫描的行数越少越好，最好只扫描需要的数据，避免扫描多余的数据；
2. 使用合适的索引，SQL 中的 WHERE 条件，需要保证命中最优的索引，索引选择错或者全表扫描，性能可能会很差。
3. 使用合适的 Join 类型，根据查询中各个表的大小和关联性，选择合适的 Join 类型（本文不讲）。
4. 使用合适的数据库。首先你要明确你的业务是数据分析型还是业务型的，根据不同业务场景选择 OLTP、OLAP 数据库（本文不讲）。

### 详细案例

案例分析开始前，准备一些数据

数据库：TIDB，TIDB 兼容 MYSQL 协议的，没用过 TIDB 不影响本文阅读；

数据表：books，图书系统核心表；

数据量：37w。

```
drop table if exists books;
create table books
(
    id          bigint          null,
    name        varchar(255)    null,
    title       varchar(255)    null,
    create_time bigint          null,
    update_time bigint          null,
    price       DECIMAL(15, 2) NOT NULL DEFAULT 0.00
);
```

日常开发中，不管上层业务如何编排、业务复杂 or 简单，对业务表的操作无非就是 CURD 只是 SQL 复杂程度不同而已，每种操作 SQL 都可能存在性能问题导致数据库稳定性。

#### DQL 最佳实践（SELECT）

对于查询类 SQL ，能单表出结果就不要 JOIN，尤其是 JOIN 多表，数据体量大、索引多的情况下数据库索引选错的概率非常大。另外，统计业务尽量不要走关系型数据库。

查询类 SQL 一般遇到下面这些性能问题

##### 全表扫描

SELECT 语句执行是全表扫描（一般是没走索引）或者是用了不合适的索引。

```
select * from books where title='java 开发实战';
```

```
```

```
+------------------+---------+--------+-------------+-------------+------+
|id                |name     |title   |create_time  |update_time  |price |
+------------------+---------+--------+-------------+-------------+------+
|1772467546674778112|未开启凭证完成任务1|java 开发实战
|1711424168639|1716288691077|102.50|
+------------------+---------+--------+-------------+-------------+------+
1 rows retrieved in 536 ms (execution: 513 ms, fetching: 23 ms)
```

上面 SQL 查询结果 1 条数据，耗时 536 ms。为什么会这么慢？看看执行计划

```
explain  analyze select * from books where title='java 开发实战';
```




从执行计划中的 TableFullScan_5 可以看出 TiDB 对表 books 进行全表扫描，然后对每一行都判断 title 是否满足条件。TableFullScan_5 的 estRows 值为 376369.00，说明扫了 376369.00 行数据，最终找到 1 行结果，耗时 536 ms，有人会说这个耗时能接受，但数据体量增加 10 倍、100 倍，SQL 性能就会非常慢了，注意：有同学可能重复多次执行 SQL 发现耗时明显降低了，这种情况是数据库缓存。

从执行计划看出 SQL 没有命中任何索引，先给 books title 列增加索引

```
CREATE INDEX idx_title ON books (title);
```

再执行 SQL 看效果

```
explain  analyze select * from books where title='java 开发实战';
```


```
1 row retrieved starting from 1 in 270 ms (execution: 249 ms, fetching: 21 ms)
```

SQL 命中了 索引 idx_title ，其 estRows 值为 90.18，说明只会扫描 90.18 行数据（数据很神奇有小数点），远远小于之前全表扫的 376369.00 行数据，所以性能明显提升了。

IndexRangeScan_8 idx_index 索引获取符合条件的索引数据，然后 TableRowIDScan_9 根据索引数据 Row ID 回表查询相应的行数据，这里回表查了数据。

如何解决回表的问题？一般使用覆盖索引。

##### 覆盖索引

上面用索引优化了全表扫描，先通过索引 idx_title 查询到符合索引数据，再通过索引的 Row ID 回表查询响应的数据，如果我们能通过索引查到最终结果减少回表数据，也能大大提高 SQL 效率。

假设业务上通过 title 查询 price 和 id，SQL 如下

```
select id,title,price from books where title='java 开发实战';
```

```
+-------------------+---------+------+
|id                 |title    |price |
+-------------------+---------+------+
|17724675466747781 12|java 开发实战|102.50|
+-------------------+---------+------+
1 row retrieved starting from 1 in 270 ms (execution: 249 ms, fetching: 21 ms)
```

索引只包含了 title 信息，所以需要回表查询 id 和 price 数据。下面我们删除 idx_title 索引，新增 id、title、price 列组合索引

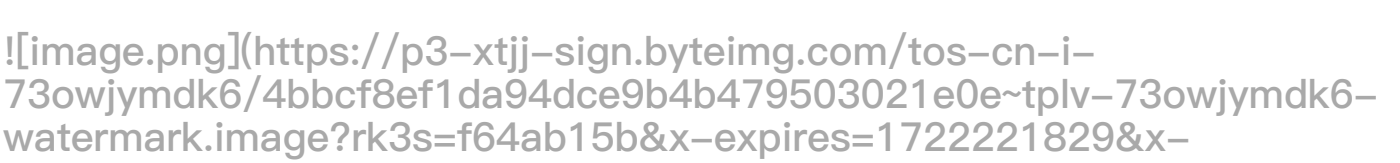
```
drop  index  idx_title on books;
CREATE INDEX idx_title_id_price ON books (title,id,price);
```

执行 SQL

```
1 row retrieved starting from 1 in 204 ms (execution: 178 ms, fetching: 26 ms)
```

增加了 idx_title_id_price 索引后，SQL 执行效率更高，速度更快，查看执行计划

```
explain  analyze select id,title,price from books where title='java 开发实战';
```



signature=H45UGrzXD2W2eURu%2Bc%2BIjbSxXEs%3D)

执行计划可以看出从索引中查询了最终结果，并没有回表了。

覆盖索引其实并非最快的，还有一种更快的方式，通过主键点查。

##### 主键点查

先给 id 列增加主键索引。

```
ALTER TABLE books ADD PRIMARY KEY(id);
```

执行 SQL

```
select * from books where id=17724675466747781l2;
```

执行结果

```
+------------------+----------+---------+-------------+-------------+------+
|id                |name      |title    |create_time  |update_time  |price |
+------------------+----------+---------+-------------+-------------+------+
|17724675466747781l2|未开启凭证完成任务1|java 开发实战
|1711424168639|1716288691077|102.50|
+------------------+----------+---------+-------------+-------------+------+
1 row retrieved starting from 1 in 199 ms (execution: 178 ms, fetching:
21 ms)
```

从结果来看，执行效率比覆盖索引更高、更快，查看执行计划

```
explain analyze  select * from books where id=17724675466747781l2;
```

从执行计划看出，Point_Get（又名“点查”），执行速度也非常快。

本文暂时不讲 join 类执行计划，join 类 SQL 更复杂，放到下一节专门来讲。在日常开发中，尽量少用 join 查询尤其是大表场景，用不好很容易慢查询，可以从单表查询结果后，再 byids 点查询附表数据即可，这块我踩了非常多坑的。亿级表优化思路之 SQL 篇，值得收藏 - 掘金

##### 避免不必要的信息

如果非必要，不要总用 select * 返回所有列数据，日常开发中发现一些同学为了简便会把一些无用的数据带出来，然后用代码过滤，这些完全没必要。

```
select * from books where title='java 开发实战';
```

改为

```
select id,title,price from books where title='java 开发实战';
```

业务上可以做一些规则，根据外部传入的列信息查询时返回对应的列即可（默

认返回一些基础列），业务方调用会复杂一些，针对列多的场景收益是非常大的。

比如：默认只返回列是 id，其它列业务方传入则返回。

#### DML 最佳实践（INERT、UPDATE、DELETE）

##### 多行数据操作切勿单条 SQL 操作（攒批概念）

当需要修改多行数据时，推荐使用单个 SQL 操作多行数据的语句，避免单条 SQL 处理。所以，业务层一定要做好赞批操作。

```
-- 不推荐做法
INSERT INTO books VALUES (1, 'GO 开发实战');
INSERT INTO books VALUES (2, 'C++ 开发实战');

DELETE FROM books WHERE id = 1;
DELETE FROM books WHERE id = 2;

-- 推荐做法
INSERT INTO books VALUES (1, 'GO 开发实战'), (2, 'C++ 开发实战');
DELETE FROM books WHERE id IN (1, 2);
```

##### 删除数据

删除数据应该遵循下面规则

1. 删除语句中指定 WHERE 条件，考虑安全层面和性能层面；
2. 如果删除大量行(数万或更多)的时候，使用批量删除，对于分布式数据库是有事务限制的；
3. 如果删除表内的所有数据，不要使用 DELETE 语句，应该使用 TRUNCATE 语句；
4. 删除数据也是一次检索数据的过程，检索符合条件的数据删除，一定要确保 WHERE 条件正确命中索引；
5. 如果需要删除表内的所有数据，不要使用 DELETE 语句，应该使用 TRUNCATE 语句。

#####

###### TRUNCATE 代替 DELETE

当一个表数据不需要时需要删除全表数据，应该使用 truncate 或者 drop（如果确定表不要了可以用drop）而非使用 delete 。

```
truncate books;
drop table books;
```

不推荐使用

```
delete books;
```

###### 批量删除数据

假设某一客户不需要某段时间数据，需要删除某段时间数据，删除 SQL 如下

```
DELETE FROM books WHERE create_time >= ’1709890312467’ AND create_time <= ’1720701398113’;
```

SQL 看上去没什么问题，但在数据体量大场景，比如 10w 甚至上 100w 场景上面 SQL 就不适合了。分布式数据库，一般都有事务限制，超出事务限制数据库会向业务层抛异常，TIDB 允许单次批量删除 10000 行数据（这个数字并非准确的，跟表的列有关系，可以自己测试一个最佳数字）。删除数据规则如下

1. 待删除数据行数<=10000，用上面删除 SQL 没问题；
2. 待删除数据行数>10000，采用循环删除，直到删除数据<10000 时退出。

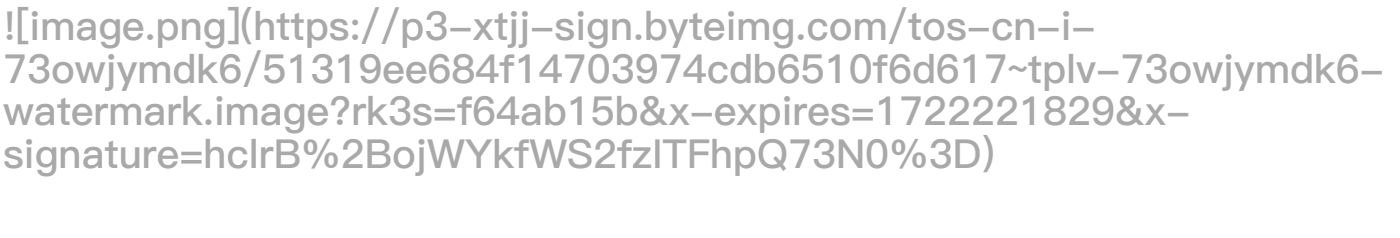
```
affectedRows := int64(-1)
for affectedRows<10000{
   affectedRows =  DELETE FROM books WHERE create_time >= '1709890312467' AND  create_time <= '1720701398113' limit 10000;
}
```

上面的代码是我偷懒写的看懂就行，在业务逻辑层 for 循环删除即可。这里需要注意，如果数据体量大的场景，建议大家做异步删除（比如每次删除投递队列，下次消费再删除），避免同步删除大量数据导致业务方逻辑被阻塞（生产环境踩了不少坑）。

各位是否发现删除数据我并没有解释执行计划，因为 delete 语法并不支持 explain analyze。但可以把 delete 语句转换成 select 分析执行计划也是一样。

```
explain analyze  select title ,price FROM books WHERE create_time >= '1709890312467' AND  create_time <= '1720701398113' limit 1000;
```

delete 语句转换成 select 语句后执行计划如下




从执行计划看出全表扫描了，需要给 create_time 列增加索引。

##### 更新数据

update 用于修改指定表数据，和 delete、select 相似，更新表数据也需要遵循一些规则

1. 在更新语句中指定 WHERE 条件，必要时指定 limit 条数。
2. 需要更新大量行(数万或更多)的时候，使用批量更新，对于分布式数据库是有事务限制的；另外，一次性过多的数据更新，将导致持有锁时间过长（悲观事务），或产生大量冲突（乐观事务）。

关于 update 上篇文章已经讲过了，可以翻翻看上亿表查询、大批量数据更新优化 - 掘金

update 语法也并不支持 explain analyze，可以把 SQL 语句转换为 select 分析执行计划，参考 delete 。

#### 索引最佳实践

索引能提高 SQL 执行效率，索引的创建和使用也要遵循一些规则

##### 创建索引规则

1. 不需要的索引及时删除，避免创建不需要的索引，新增一个索引是有代价的。每增加一个索引，在插入一条数据的时，就要存储索引数据。索引越多，写入越慢、并且空间占用越大。另外过多索引会影响 SQL 优化器执行时间，索引过多可能会误导优化器选择错误索引，尤其是大体量表非常明显；
2. 创建索引考虑查询能用上覆盖索引大幅度提升性能。这里敲黑板，日常开发中是很难把控的，比如业务需要 10 个列，但建索引不可能把 10 个列都加上，所以并不是所有场景都适合，所以不要盲目的使用覆盖索引；
3. 原则上对查询中需要用到的列创建索引，目的是提高性能。但有些情况并不适合

a. 创建索引选择过滤性好的列，通过增加索引可以显著提高过滤后的行数，比如身份证号码、能标识唯一性的等，但有一些列不合适，比如状态、性别这类过滤性太弱了，比如在 books 表新增一个布尔类型字段 is_del 代表改行是否删除，查询语句如下：

```
select title ,price FROM books WHERE title ='java 开发实战' and is_del = false;
-- 创建索引时并不需要加 is_del 过滤性太弱加上并没有意义
CREATE INDEX idx_title_create_time ON books (title);
```

```

b.where 跟多个查询条件时，可以选择组合索引，把等值条件的列放在组合索引的前面，比如：

```
select title ,price FROM books WHERE title =’java 开发实战’ and create_time >= ’1709890312467’ AND  create_time <= ’1720701398113’ limit 1000;
CREATE INDEX idx_title_create_time ON books (title,create_time);
```

##### 索引使用规则

索引创建好了并不代表索引有意义，索引的目的是为了加速查询，索引使用也要遵循一些规则

1. 确保索引在一些查询中被用上，如果一个索引没有被用上，那这个索引是没有意义的，即使删除；
2. 使用索引时需要满足左前缀规则。

以 books 表为案例，假设某个业务场景需要对 id、title、price 列建索引；

```
CREATE INDEX idx_title_id_price ON books (title,id,price);
```

下面 SQL 能用上索引

```
select id,title,price from books where title=’java 开发实战’;
```

下面 SQL 不能用上索引
```

```
select id,title,price from books where price=102.50;
```

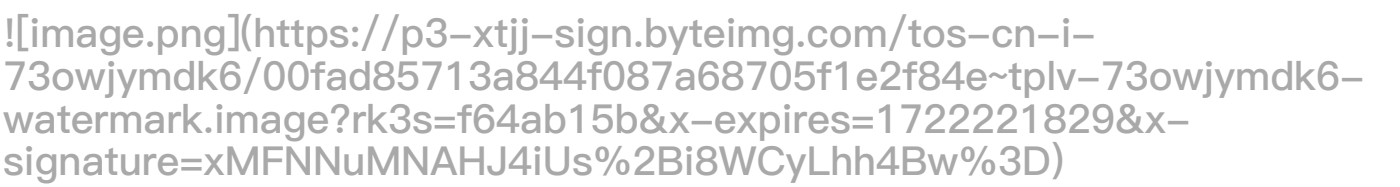
尽量使用覆盖索引（自己根据业务判断），避免使用 select * 需要业务控制；

```
select id,title,price from books where title='java 开发实战';
```

查询条件使用 !=，NOT IN 时虽然能用上索引，可能效果并不明显；

```
select id,title,price from books where title != 'java 开发实战';
select id,title,price from books where title not in ('java 开发实战');
```

分析执行计划会发现，虽然用上了索引但基本都是全表扫




使用 LIKE 时并且条件是以通配符 % 开头，也无法使用索引；

```
-- 下面 sql 全索引扫描，不推荐
select id,title,price from books where title like '%java 开发实战%';

-- 下面 sql 正确命中索引
select id,title,price from books where title like 'java 开发实战%';
```

查询条件是 in 时，建议不要超过 500 个；

当有多个索引提供使用，SQL 优化器索引选择错误，你知道最优索引时建议使用强制索引。

### 总结

SQL 调优其实非常宽泛，并不只包含查询类 SQL 、索引优化等。当被问如何做 SQL 调优可以从上文角度全方位给提问者剖析，你应该会得到他的认可。

原文链接: https://juejin.cn/post/7390646355028541467