32 | MySQL调优之SQL语句：如何写出高性能SQL语句？

2019-08-06 刘超

你好，我是刘超。

从今天开始，我将带你一起学习MySQL的性能调优。MySQL数据库是互联网公司使用最为频繁的数据库之一，不仅仅因为它开源免费，MySQL卓越的性能、稳定的服务以及活跃的社区都成就了它的核心竞争力。

我们知道，应用服务与数据库的交互主要是通过SQL语句来实现的。在开发初期，我们更加关注的是使用SQL实现业务功能，然而系统上线后，随着生产环境数据的快速增长，之前写的很多SQL语句就开始暴露出性能问题。

在这个阶段中，我们应该尽量避免一些慢SQL语句的实现。但话说回来，SQL语句慢的原因千千万，除了一些常规的慢SQL语句可以直接规避，其它的一味去规避也不是办法，我们还要学会如何去分析、定位到其根本原因，并总结一些常用的SQL调优方法，以备不时之需。

那么今天我们就重点看看慢SQL语句的几种常见诱因，从这点出发，找到最佳方法，开启高性能SQL语句的大门。

慢SQL语句的几种常见诱因

1. 无索引、索引失效导致慢查询

如果在一张几千万数据的表中以一个没有索引的列作为查询条件，大部分情况下查询会非常耗

时，这种查询毫无疑问是一个慢SQL查询。所以对于大数据量的查询，我们需要建立适合的索引来优化查询。

虽然我们很多时候建立了索引，但在一些特定的场景下，索引还有可能会失效，所以索引失效也是导致慢查询的主要原因之一。针对这点的调优，我会在第34讲中详解。

2. 锁等待

我们常用的存储引擎有 InnoDB 和 MyISAM，前者支持行锁和表锁，后者只支持表锁。

如果数据库操作是基于表锁实现的，试想下，如果一张订单表在更新时，需要锁住整张表，那么其它大量数据库操作（包括查询）都将处于等待状态，这将严重影响到系统的并发性能。

这时，InnoDB 存储引擎支持的行锁更适合高并发场景。但在使用 InnoDB 存储引擎时，我们要特别注意行锁升级为表锁的可能。在批量更新操作时，行锁就很可能会升级为表锁。

MySQL认为如果对一张表使用大量行锁，会导致事务执行效率下降，从而可能造成其它事务长时间锁等待和更多的锁冲突问题发生，致使性能严重下降，所以MySQL会将行锁升级为表锁。还有，行锁是基于索引加的锁，如果我们在更新操作时，条件索引失效，那么行锁也会升级为表锁。

因此，基于表锁的数据库操作，会导致SQL阻塞等待，从而影响执行速度。在一些更新操作（insert\update\delete）大于或等于读操作的情况下，MySQL不建议使用MyISAM存储引擎。

除了锁升级之外，行锁相对表锁来说，虽然粒度更细，并发能力提升了，但也带来了新的问题，那就是死锁。因此，在使用行锁时，我们要注意避免死锁。关于死锁，我还会在第35讲中详解。

3. 不恰当的SQL语句

使用不恰当的SQL语句也是慢SQL最常见的诱因之一。例如，习惯使用<SELECT *>，<SELECT COUNT(*)> SQL语句，在大数据表中使用<LIMIT M,N>分页查询，以及对非索引字段进行排序等等。

优化SQL语句的步骤

通常，我们在执行一条SQL语句时，要想知道这个SQL先后查询了哪些表，是否使用了索引，这些数据从哪里获取到，获取到数据遍历了多少行数据等等，我们可以通过EXPLAIN命令来查看这些执行信息。这些执行信息被统称为执行计划。

1. 通过EXPLAIN分析SQL执行计划

假设现在我们使用EXPLAIN命令查看当前SQL是否使用了索引，先通过SQL EXPLAIN导出相应的执行计划如下：

```
EXPLAIN SELECT * FROM `order` where id < 10;
```

信息 Result 1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ALL	PRIMARY	(Null)	(Null)	(Null)	581	33.33	Using whe

下面对图示中的每一个字段进行一个说明，从中你也能收获到很多零散的知识点。

- id：每个执行计划都有一个id，如果是一个联合查询，这里还将有多个id。
- select_type：表示SELECT查询类型，常见的有SIMPLE（普通查询，即没有联合查询、子查询）、PRIMARY（主查询）、UNION（UNION中后面的查询）、SUBQUERY（子查询）等。
- table：当前执行计划查询的表，如果给表起别名了，则显示别名信息。
- partitions：访问的分区表信息。
- type：表示从表中查询到行所执行的方式，查询方式是SQL优化中一个很重要的指标，结果值从好到差依次是：system > const > eq_ref > ref > range > index > ALL。

```
EXPLAIN SELECT * FROM `order` where id = 2;
```

信息 Result 1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	const	PRIMARY	PRIMARY	4	const	1	100	(Null)

- system/const：表中只有一行数据匹配，此时根据索引查询一次就能找到对应的数据。如果是B +树索引，我们知道此时索引构造成了多个层级的树，当查询的索引在树的底层时，查询效率就越低。const表示此时索引在第一层，只需访问一层便能得到数据。

```
EXPLAIN SELECT * FROM `order` a, order_detail b where a.id = b.order_id;
```

信息 Result 1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	b	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	1	100	Using where
1	SIMPLE	a	(Null)	eq_ref	PRIMARY	PRIMARY	4	demo.b.or	1	100	(Null)

- eq_ref：使用唯一索引扫描，常见于多表连接中使用主键和唯一索引作为关联条件。

```
EXPLAIN SELECT * FROM `order` where order_no = "2";
```

信息 Result 1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ref	idx_order	idx_order	5	const	1	100	(Null)

- ref：非唯一索引扫描，还可见于唯一索引最左原则匹配扫描。

```
EXPLAIN SELECT * FROM `order` where id > 1;
```

信息 Result 1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	range	PRIMARY	PRIMARY	4	(Null)	580	100	Using where

- range：索引范围扫描，比如，<，>，between等操作。

```
1  EXPLAIN SELECT id FROM `order`;
2
3
4
```

信息 | Result 1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	index	(Null)	idx_order	5	(Null)	581	100	Using index

- index：索引全表扫描，此时遍历整个索引树。

```
1  EXPLAIN SELECT * FROM `order` where pay_money = 0;
2
3
4
```

信息 | Result 1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	order	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	581	10	Using where

- ALL：表示全表扫描，需要遍历全表来找到对应的行。
- possible_keys：可能使用到的索引。
- key：实际使用到的索引。
- key_len：当前使用的索引的长度。
- ref：关联id等信息。
- rows：查找到记录所扫描的行数。
- filtered：查找到所需记录占总扫描记录数的比例。
- Extra：额外的信息。

2. 通过Show Profile分析SQL执行性能

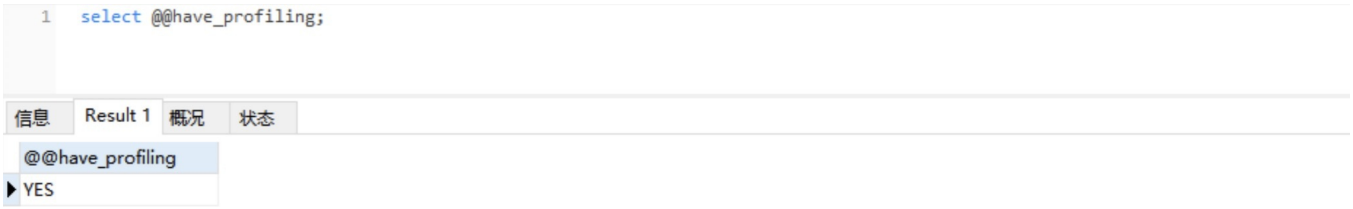
上述通过 EXPLAIN 分析执行计划，仅仅是停留在分析SQL的外部的执行情况，如果我们想要深入到MySQL内核中，从执行线程的状态和时间来分析的话，这个时候我们就可以选择Profile。

Profile除了可以分析执行线程的状态和时间，还支持进一步选择ALL、CPU、MEMORY、BLOCK IO、CONTEXT SWITCHES等类型来查询SQL语句在不同系统资源上所消耗的时间。以下是相关命令的注释：

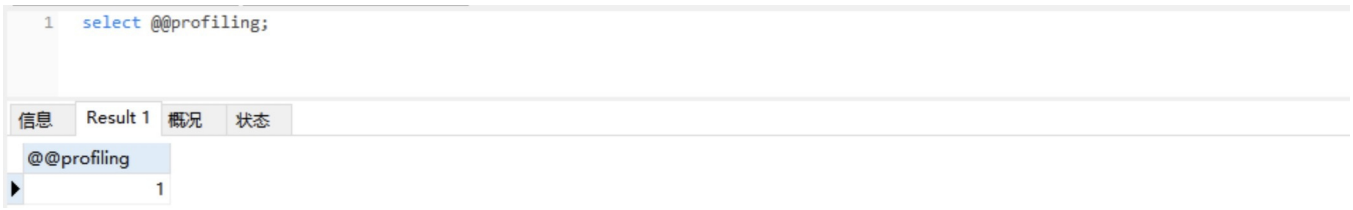
```
SHOW PROFILE [type [, type] ... ]
[FOR QUERY n]
[LIMIT row_count [OFFSET offset]]

type参数：
| ALL：显示所有开销信息
| BLOCK IO：阻塞的输入输出次数
| CONTEXT SWITCHES：上下文切换相关开销信息
| CPU：显示CPU的相关开销信息
| IPC：接收和发送消息的相关开销信息
| MEMORY ：显示内存相关的开销，目前无用
| PAGE FAULTS ：显示页面错误相关开销信息
| SOURCE ：列出相应操作对应的函数名及其在源码中的调用位置(行数)
| SWAPS：显示swap交换次数的相关开销信息
```

值得注意的是，MySQL是在5.0.37版本之后才支持Show Profile功能的，如果你不太确定的话，可以通过select @@have_profiling查询是否支持该功能，如下图所示：



最新的MySQL版本是默认开启Show Profile功能的，但在之前的旧版本中是默认关闭该功能的，你可以通过set语句在Session级别开启该功能：



Show Profiles只显示最近发给服务器的SQL语句，默认情况下是记录最近已执行的15条记录，我们可以重新设置profiling_history_size增大该存储记录，最大值为100。

```
1  SELECT COUNT(*) FROM `order`;
2  show profiles;
```

信息 | Result 1 | 概况 | 状态

Query_ID	Duration	Query
1161	0.002482	SHOW STATUS
1162	0.0020535	SHOW STATUS
1163	0.001888	SHOW STATUS
1164	0.0019845	SELECT QUERY_ID, SUM(DURATION) AS SUM_DURATI
1165	0.00190325	SELECT STATE AS `Status`, ROUND(SUM(DURATION),7
1166	0.000205	SET PROFILING = 1
1167	0.001532	SHOW STATUS
1168	0.0014975	SHOW STATUS
1169	0.00044325	SELECT COUNT(*) FROM `order`
1170	0.00127325	SHOW STATUS
1171	0.00124975	SELECT QUERY_ID, SUM(DURATION) AS SUM_DURATI
1172	0.0010205	SELECT STATE AS `Status`, ROUND(SUM(DURATION),7
1173	0.0003905	SET PROFILING = 1
1174	0.00254275	SHOW STATUS
1175	0.001943	SHOW STATUS

获取到Query_ID之后，我们再通过Show Profile for Query ID语句，就能够查看到对应Query_ID的SQL语句在执行过程中线程的每个状态所消耗的时间了：

```
1  show profile for query 1198;
```

信息 | Result 1 | 概况 | 状态

Status	Duration
starting	0.00009
checking permissions	0.000011
Opening tables	0.000031
init	0.000011
System lock	0.000014
optimizing	0.000009
statistics	0.00002
preparing	0.000017
executing	0.000006
Sending data	0.000053
end	0.000008
query end	0.000007
waiting for handler comm	0.00001
query end	0.000009
closing tables	0.000011
freeing items	0.000052
cleaning up	0.000016

通过以上分析可知：SELECT COUNT(*) FROM `order`; SQL语句在Sending data状态所消耗的时间最长，这是因为在该状态下，MySQL线程开始读取数据并返回到客户端，此时有大量磁盘I/O操作。

常用的SQL优化

在使用一些常规的SQL时，如果我们通过一些方法和技巧来优化这些SQL的实现，在性能上就会比使用常规通用的实现方式更加优越，甚至可以将SQL语句的性能提升到另一个数量级。

1. 优化分页查询

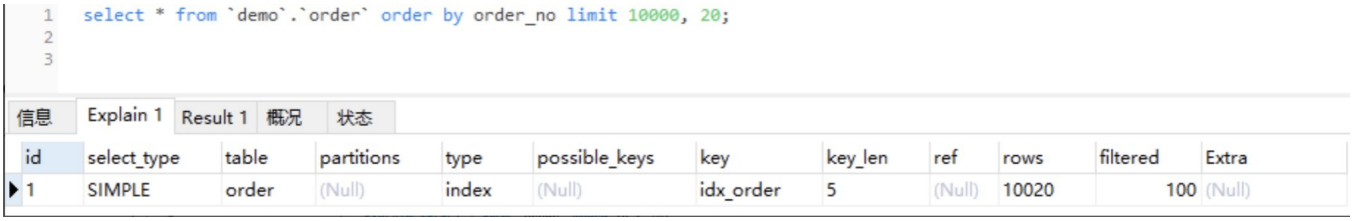
通常我们是使用<LIMIT M,N> +合适的order by来实现分页查询，这种实现方式在没有任何索引条件支持的情况下，需要做大量的文件排序操作（file sort），性能将会非常得糟糕。如果有对应的索引，通常刚开始的分页查询效率会比较理想，但越往后，分页查询的性能就越差。

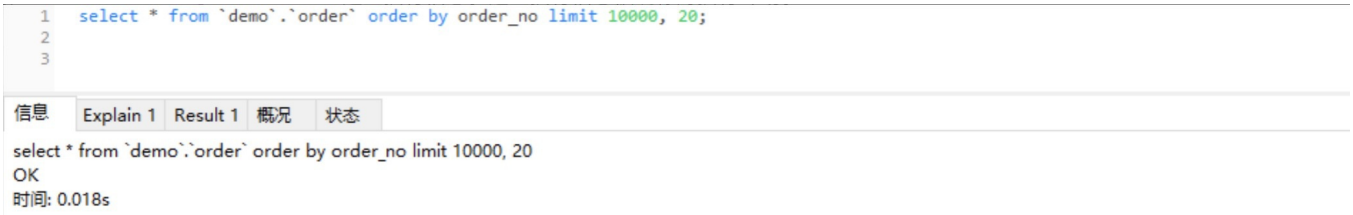
这是因为我们在使用LIMIT的时候，偏移量M在分页越靠后的时候，值就越大，数据库检索的数据也就越多。例如 LIMIT 10000,10这样的查询，数据库需要查询10010条记录，最后返回10条记录。也就是说将会有10000条记录被查询出来没有被使用到。

我们模拟一张10万数量级的order表，进行以下分页查询：

```
select * from `demo`.`order` order by order_no limit 10000, 20;
```

通过EXPLAIN分析可知：该查询使用到了索引，扫描行数为10020行，但所用查询时间为0.018s，相对来说时间偏长了。



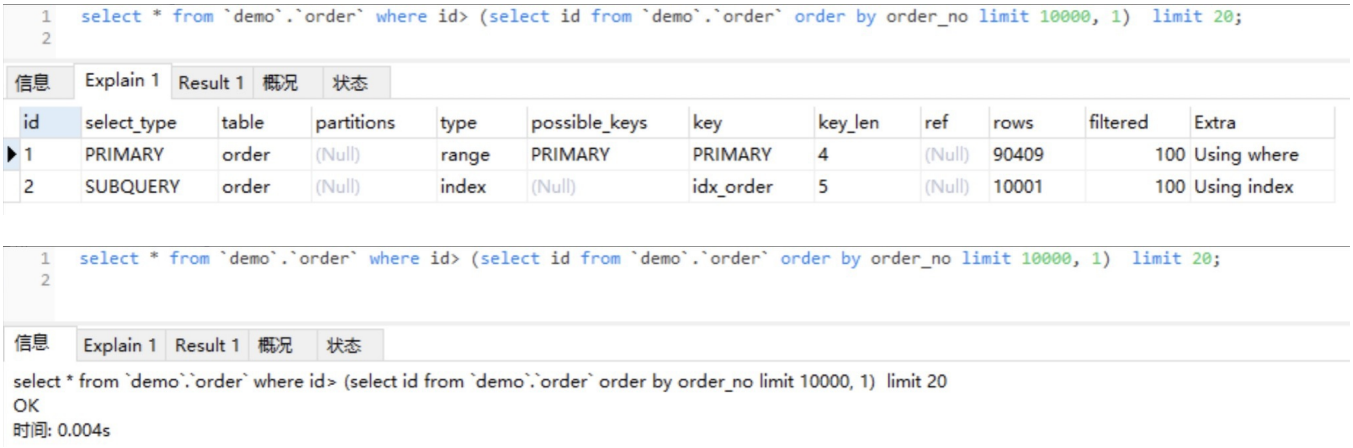


- 利用子查询优化分页查询

以上分页查询的问题在于，我们查询获取的10020行数据结果都返回给我们了，我们能否先查询出所需要的20行数据中的最小ID值，然后通过偏移量返回所需要的20行数据给我们呢？我们可以通过索引覆盖扫描，使用子查询的方式来实现分页查询：

```
select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1)  limit 20;
```

通过EXPLAIN分析可知：子查询遍历索引的范围跟上一个查询差不多，而主查询扫描了更多的行数，但执行时间却减少了，只有0.004s。这就是因为返回行数只有20行了，执行效率得到了明显的提升。



```
select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1)  limit 20;
```

信息 | Explain 1 | Result 1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	order	(Null)	range	PRIMARY	PRIMARY	4	(Null)	90409	100	Using where
2	SUBQUERY	order	(Null)	index	(Null)	idx_order	5	(Null)	10001	100	Using index


```
select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1)  limit 20;
```

信息 | Explain 1 | Result 1 | 概况 | 状态

```
select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1)  limit 20
OK
时间: 0.004s
```

2. 优化SELECT COUNT(*)

COUNT()是一个聚合函数，主要用来统计行数，有时候也用来统计某一列的行数量（不统计NULL值的行）。我们平时最常用的就是COUNT(*)和COUNT(1)这两种方式了，其实两者没有明显的区别，在拥有主键的情况下，它们都是利用主键列实现了行数的统计。

但COUNT()函数在MyISAM和InnoDB存储引擎所执行的原理是不一样的，通常在没有任何查询条件下的COUNT(*)，MyISAM的查询速度要明显快于InnoDB。

这是因为MyISAM存储引擎记录的是整个表的行数，在COUNT(*)查询操作时无需遍历表计算，直接获取该值即可。而在InnoDB存储引擎中就需要扫描表来统计具体的行数。而当带上where条件语句之后，MyISAM跟InnoDB就没有区别了，它们都需要扫描表来进行行数的统计。

如果对一张大表经常做SELECT COUNT(*)操作，这肯定是不明智的。那么我们该如何对大表的COUNT()进行优化呢？

- 使用近似值

有时候某些业务场景并不需要返回一个精确的COUNT值，此时我们可以使用近似值来代替。我们可以使用EXPLAIN对表进行估算，要知道，执行EXPLAIN并不会真正去执行查询，而是返回一个估算的近似值。

- 增加汇总统计

如果需要一个精确的COUNT值，我们可以额外新增一个汇总统计表或者缓存字段来统计需要的COUNT值，这种方式在新增和删除时有一定的成本，但却可以大大提升COUNT()的性能。

3. 优化SELECT *

我曾经看过很多同事习惯在只查询一两个字段时，都使用select * from table where xxx这样的SQL语句，这种写法在特定的环境下会存在一定的性能损耗。

MySQL常用的存储引擎有MyISAM和InnoDB，其中InnoDB在默认创建主键时会创建主键索引，而主键索引属于聚簇索引，即在存储数据时，索引是基于B +树构成的，具体的行数据则存储在叶子节点。

而MyISAM默认创建的主键索引、二级索引以及InnoDB的二级索引都属于非聚簇索引，即在存储数据时，索引是基于B +树构成的，而叶子节点存储的是主键值。

假设我们的订单表是基于InnoDB存储引擎创建的，且存在order_no、status两列组成的组合索引。此时，我们需要根据订单号查询一张订单表的status，如果我们使用select * from order where order_no='xxx'来查询，则先会查询组合索引，通过组合索引获取到主键ID，再通过主键ID去主键索引中获取对应行所有列的值。

如果我们使用select order_no, status from order where order_no='xxx'来查询，则只会查询组合索引，通过组合索引获取到对应的order_no和status的值。如果你对这些索引还不够熟悉，请重点关注之后的第34讲，那一讲会详述数据库索引的相关内容。

总结

在开发中，我们要尽量写出高性能的SQL语句，但也无法避免一些慢SQL语句的出现，或因为疏漏，或因为实际生产环境与开发环境有所区别，这些都是诱因。面对这种情况，我们可以打开慢SQL配置项，记录下都有哪些SQL超过了预期的最大执行时间。首先，我们可以通过以下命令行查询是否开启了记录慢SQL的功能，以及最大的执行时间是多少：

```
Show variables like 'slow_query%';
Show variables like 'long_query_time';
```

如果没有开启，我们可以通过以下设置来开启：

```
set global slow_query_log='ON'; //开启慢SQL日志
set global slow_query_log_file='/var/lib/mysql/test-slow.log';//记录日志地址
set global long_query_time=1;//最大执行时间
```

除此之外，很多数据库连接池中间件也有分析慢SQL的功能。总之，我们要在编程中避免低性能的SQL操作出现，除了要具备一些常用的SQL优化技巧之外，还要充分利用一些SQL工具，实现SQL性能分析与监控。

思考题

假设有一张订单表order，主要包含了主键订单编码order_no、订单状态status、提交时间create_time等列，并且创建了status列索引和create_time列索引。此时通过创建时间降序获取状态为1的订单编码，以下是具体实现代码：

```
select order_no from order where status =1 order by create_time desc
```

你知道其中的问题所在吗？我们又该如何优化？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

张学磊

14

status和create_time单独建索引，在查询时只会遍历status索引对数据进行过滤，不会用到create_time列索引，将符合条件的数据返回到server层，在server对数据通过快排算法进行排序，Extra列会出现file sort；应该利用索引的有序性，在status和create_time列建立联合索引，这样根据status过滤后的数据就是按照create_time排好序的，避免在server层排序

2019-08-06

作者回复

非常准确！

2019-08-06

QQ怪 2

对staus和create_time建立联合索引

2019-08-06

作者回复

对的，为了避免文件排序的发生。因为查询时我们只能用到status索引，如果要对create_time进行排序，则需要使用文件排序filesort。

filesort是通过相应的排序算法将取得的数据在内存中进行排序，如果内存不够则会使用磁盘文件作为辅助。虽然在一些场景中，filesort并不是特别消耗性能，但是我们可以避免filesort就尽量避免。

2019-08-06

迎风劲草 2

创建 status create_time order_no 联合索引，避免回表

2019-08-06

作者回复

建立联合索引没错，还有就是避免文件排序的问题。

2019-08-06

Kian.Lee 1

我在实际项目中使用“select order_no from order where status =1 order by id desc ”代替此功能，id为bigint，也少维护一个索引（create_time）

2019-08-08

Jian 0

因为好久没有做SQL相关的开发了，所以开始没有特别明白【利用子查询优化分页查询】这里面的意思。我来说下自己的想法，请您检证。我看到您贴的截图中，优化后的sql语句，扫描的行数（rows列）分别是90409和10001，多余前一个较慢的查询，可见扫描行数，不是这个性能的主要原因。我推测这个是由于limit [m],n的实现方法导致的，即MySql会把m+n的数据都取出来，然后返回n个数据给用户。如果用第二种SQL语句，子查询只是获得一个id，虽然扫描了很多行，但都是在索引上进行的，切不需要回表获取内容。外查询是根据id获取数据20条内容，速度自然就会快了。我认为这里性能提高的原因还是居于索引的恰当使用。

2019-08-12

JackJin 0

感觉要建立联合索引，但不知具体原因

2019-08-09

作者回复

为了避免文件排序的发生。因为查询时我们只能用到status索引，如果要对create_time进行排

序，则需要使用文件排序filesort。

2019-08-12

Geek_002ff7 0

真实情况一般不会在status上单独建索引，因为status大部分都是重复值，数据库一般走全表扫描了，感觉漏讲了索引失效的情况

2019-08-09

作者回复

下一讲则会讲到

2019-08-12

东方奇骥 0

select * from `demo`.`order` order by order_no limit 10000, 20;
select * from `demo`.`order` where id> (select id from `demo`.`order` order by order_no limit 10000, 1) limit 20; 老师，感觉自己没完全弄明白，就是用子查询快那么多，但是子查询里，不是也要扫描10001行？还是说子查询里只查了id，不需要回行，所以速度快？

2019-08-08

作者回复

这个涉及到返回记录的大小，前者会返回10020条行记录，而后者只返回20条记录。

2019-08-09

LW 0

order_no创建主键，status+create_time创建联合索引

2019-08-06

作者回复

对的

2019-08-06

撒旦的堕落 0

订单状态字段的离散度很低 不适合做索引
因为离散度低 而又没有分页 所以当表数据量大的时候 查询出来的数量也有可能很大
创建时间倒序 可以换成主键倒序 去除掉时间字段的索引
根据状态查询 个人觉得可以从业务入手 将相同状态的数据保存到一张表 想听听老师的意见

2019-08-06

作者回复

这里主要是filesort问题

2019-08-06

nimil 0

select * from table limit 1 这种sql语句会走主键索引么，我看explain里边没有任何索引记录

2019-08-06

作者回复

不会，没有使用到索引。

2019-08-06

密码123456

0

有Oracle吗？oracle感觉用的人不多了，是不是要被淘汰了？

2019-08-06

门窗小二

0

创建status，创建时间及订单号联合索引，其中创建时间制定降序，这样避免产生filesort及回表！不知道是否正确？

2019-08-06

作者回复

对的，赞

2019-08-06

我知道了嗯

0

感觉订单状态不需要索引

2019-08-06

Zed

0

这里感觉有俩缺陷

1、会全表排序

2、订单状态过滤效果不佳

2019-08-06