查询优化

在编写快速的查询之前，需要清楚一点，真正重要的是响应时间，而且要知道在整个SQL语句的执行过程中每个步骤都花费了多长时间，要知道哪些步骤是拖垮执行效率的关键步骤，想要做到这点，必须要知道查询的生命周期，然后进行优化，不同的应用场景有不同的优化方式，不要一概而论，具体情况具体分析

▼ 查询慢的原因

网络

CPU

IO

上下文切换：如CPU任务切换的恢复现场和保护现场等

系统调用

生成统计信息

锁等待时间

▼ 优化数据访问

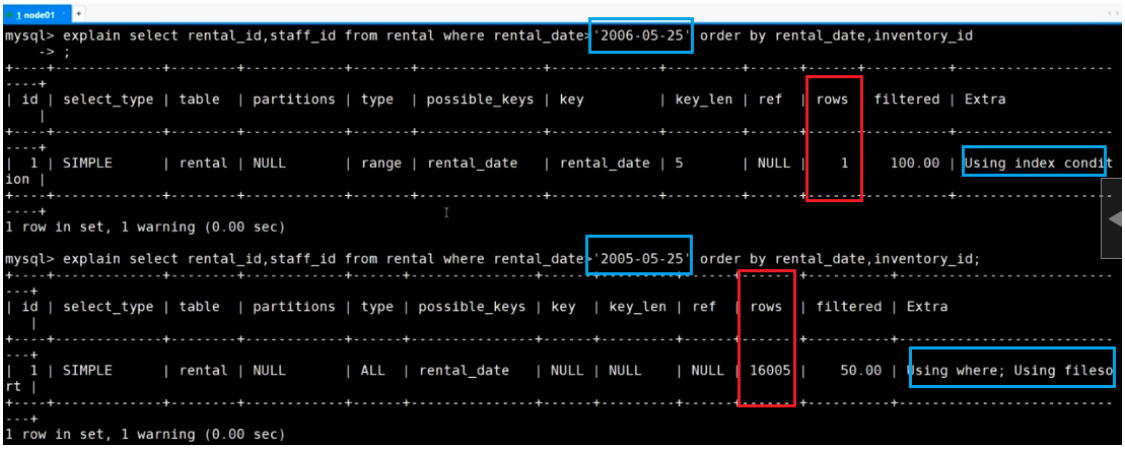
- 查询性能低下的主要原因是访问的数据太多，某些查询不可避免的需要筛选大量的数据，我们可以通过减少访问数据量的方式进行优化

  1、确认应用程序是否在检索大量超过需要的数据

  2、确认mysql服务器层是否在分析大量超过需要的数据行

  案例1：

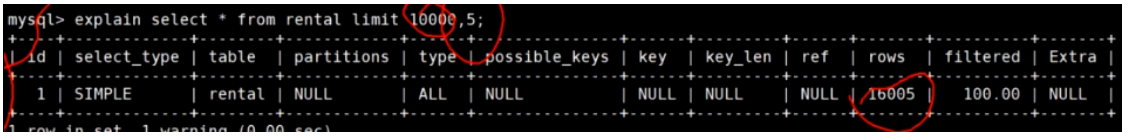
  通过修改一条sql语句的查询条件的一个数值就可能导致不会使用到索引



  结论：如果查询的结果集数据量太多的话，可能不会使用到索引

  案例2：

  使用limit 的时候 前面的数值太大，会导致全表索引，可以通过子查询进行优化操作



优化：

select * from rental a join (select rental_id from rental limit 10000,5) b on a.rental_id = b.rental_id)

- 是否向数据库请求了不需要的数据

1. 查询不需要的记录

   我们常常会误以为mysql会只返回需要的数据，实际上mysql却是先返回全部结果再进行计算，在日常的开发习惯中，经常是先用select语句查询大量的结果，然后获取前面的N行后关闭结果集。

   优化方式是在查询后面添加limit

2. 多表关联时返回全部列，不要返回不需要的数据
3. 总是取出全部列，不要使用 select *
4. 重复查询相同的数据

   如果需要不断的重复执行相同的查询，且每次返回完全相同的数据，因此，基于这样的应用场景，我们可以将这部分数据缓存起来，这样的话能够提高查询效率

▼ 执行过程的优化

- 查询缓存

  在解析一个查询语句之前，如果查询缓存是打开的，那么mysql会优先检查这个查询是否命中查询缓存中的数据，如果查询恰好命中了查询缓存，那么会在返回结果之前会检查用户权限，如果权限没有问题，那么mysql会跳过所有的阶段，就直接从缓存中拿到结果并返回给客户端——5.8之前的可以优化，5.8版本就对查询缓存器这个去掉了，因为对于经常修改的表来说缓存命中率不高，5.8之前的版本如果是一些常量表可以考虑查询缓存这个优化点

- 查询优化处理

  mysql查询完缓存之后会经过以下几个步骤：解析SQL、预处理、优化SQL执行计划，这几个步骤出现任何的错误，都可能会终止查询

1. 语法解析器和预处理

   mysql通过关键字将SQL语句进行解析，并生成一颗解析树，mysql解析器将使用mysql语法规则验证和解析查询，例如验证使用使用了错误的关键字或者顺序是否正确等等，预处理器会进一步检查解析树是否合法，例如表名和列名是否存在，是否有歧义，还会验证权限等等

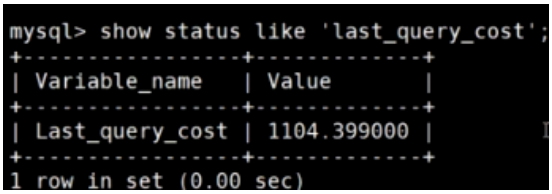
2. 查询优化器

   当语法树没有问题之后，相应的要由优化器将其转成执行计划，一条查询语句可以使用非常多的执行方式，最后都可以得到对应的结果，但是不同的执行方式带来的效率是不同的，优化器的最主要目的就是要选择最有效的执行计划

   mysql使用的是基于成本的优化器，在优化的时候会尝试预测一个查询使用某种查询计划时候的成本，并选择其中成本最小的一个

   ▼ select count(*) from film_actor;　　// 结果 5462 行

show status like 'last_query_cost'; // 最后一次查询所耗费的成本是多少
可以看到这条查询语句大概需要做1104个数据页才能找到对应的数据，这是经过一系列的统计信息计算来的



极端这个value的参数依据主要有，每个表或者索引的页面个数、索引的基数、索引和数据行的长度、索引的分布情况——根据这个参数计算的这个值不是一个准确值，是一个预估值。

▼ 在很多情况下mysql会选择错误的执行计划，原因如下：

- 统计信息不准确

  InnoDB因为其mvcc的架构，并不能维护一个数据表的行数的精确统计信息

- 执行计划的成本估算不等同于实际执行的成本

  有时候某个执行计划虽然需要读取更多的页面，但是他的成本却更小，因为如果这些页面都是顺序读或者这些页面都已经在内存中的话，那么它的访问成本将很小，mysql层面并不知道哪些页面在内存中，哪些在磁盘，所以查询之际执行过程中到底需要多少次IO是无法得知的

- mysql的最优可能跟你想的不一样

  mysql的优化是基于成本模型的优化，但是有可能不是最快的优化

- mysql不考虑其他并发执行的查询

- mysql不会考虑不受其控制的操作成本

  执行存储过程或者用户自定义函数的成本

▼ 优化器的优化策略

1. 静态优化，直接对解析树进行分析，并完成优化
2. 动态优化，动态优化与查询的上下文有关，也可能跟取值、索引对应的行数有关
3. mysql对查询的静态优化只需要一次，但对动态优化在每次执行时都需要重新评估

▼ 优化器的优化类型

1. 重新定义关联表的顺序

   数据表的关联并不总是按照在查询中指定的顺序进行，决定关联顺序时优化器很重要的功能

2. 将外连接转化成内连接，内连接的效率要高于外连接

   内连接获取的数据量少于外连接

3. 使用等价变换规则，mysql可以使用一些等价变化来简化并规划表达式

   如 a > 4 or a < 4 可以替换成 a ≠ 4　然后进行测试，选择

   这个点 影响效率有时候不是很高，因为CPU计算速度太快

4. 优化count(),min(),max()

   索引和列是否可以为空通常可以帮助mysql优化这类表达式：例如，要找到某一列的最小值，只需要查询索引的最左端的记录即可，不需要全文扫描比较

5. 预估并转化为常数表达式，当mysql检测到一个表达式可以转化为常数的时候，就会一直把该表达式作为常数进行处理

   explain select film.film_id,film_actor.actor_id from film inner join film_actor using(film_id) where film.film_id = 1

6. 索引覆盖扫描，当索引中的列包含所有查询中需要使用的列的时候，可以使用覆盖索引

7. 子查询优化

   mysql在某些情况下可以将子查询转换一种效率更高的形式，从而减少多个查询多次对数据进行访问，例如将经常查询的数据放入到缓存中

8. 等值传播

   如果两个列的值通过等式关联，那么mysql能够把其中一个列的where条件传递到另一个上：

   explain select film.film_id from film inner join film_actor using(film_id) where film.film_id > 500;

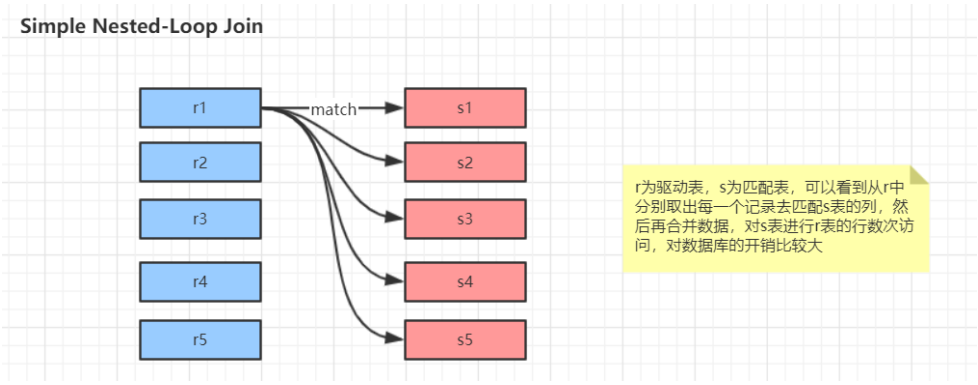
   这里使用film_id字段进行等值关联，film_id这个列不仅适用于film表而且适用于film_actor表

   explain select film.film_id from film inner join film_actor using(film_id) where film.film_id > 500 and film_actor.film_id > 500; 不需要这个，等值传播，主要是认识这个名词

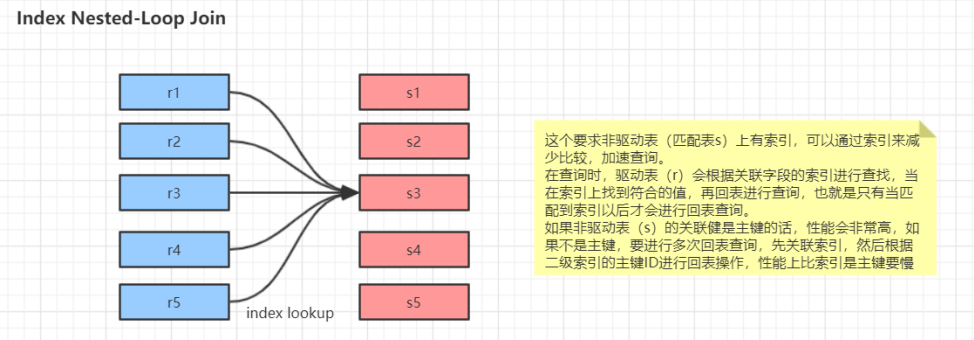
▼ 关联查询

- join的实现方式原理

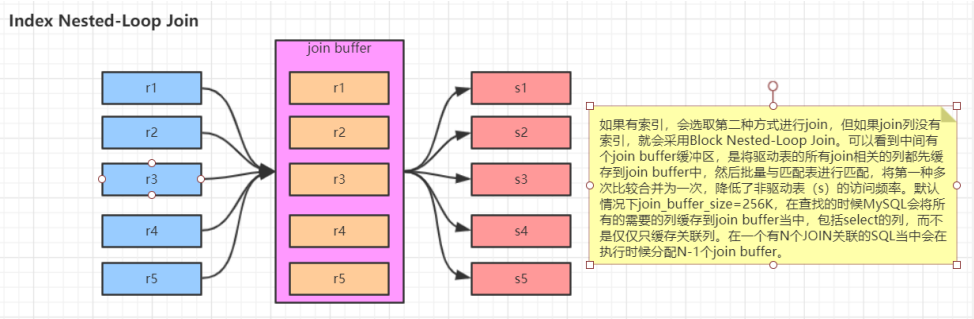
  1. Simple Nested-Loop Join



  2. Index Nested-Loop Join



3. Block Nested-Loop Join



（1）Join Buffer会缓存所有参与查询的列而不是只有Join的列。

（2）可以通过调整join_buffer_size缓存大小

（3）join_buffer_size的默认值是256K，join_buffer_size的最大值在MySQL 5.1.22版本前是4G-1，而之后的版本才能在64位操作系统下申请大于4G的Join Buffer空间。

（4）使用Block Nested-Loop Join算法需要开启优化器管理配置的optimizer_switch的设置block_nested_loop为on，默认为开启。

show variables like '%optimizer_switch%' ——- 查看优化器开关命令

- 案例演示

  查看不同的顺序执行方式对查询性能的影响：

  explain select film.film_id,film.title,film.release_year,actor.actor_id,actor.first_name,actor.last_name from film inner join f

  ilm_actor using(film_id) inner join actor using(actor_id);

  查看执行的成本：

  show status like 'last_query_cost';

  按照自己预想的规定顺序执行：

  explain select straight_join film.film_id, film.title,film.release_year, actor.actor_id, actor.first_name, actor.last_name from film inner join film_actor using(film_id) inner join actor using(actor_id);

  查看执行的成本：

show status like 'last_query_cost';

这个案例的结果：通过straight_join 固定表执行顺序，不按照mysql执行器优化的方式，结果是mysql 优化器的方式更合理，所以一般情况下不对优化器进行优化操作

▼ 排序优化

- 两次传输排序

  第一次数据读取是将需要排序的字段读取出来，然后进行排序，第二次是将排好序的结果按照需要去读取数据行。

  这种方式效率比较低，原因是第二次读取数据的时候因为已经排好序，需要去读取所有记录而此时更多的是随机IO，读取数据成本会比较高

  两次传输的优势，在排序的时候存储尽可能少的数据，让排序缓冲区可以尽可能多的容纳行数来进行排序操作

- 单次传输排序

  先读取查询所需要的所有列，然后再根据给定列进行排序，最后直接返回排序结果，此方式只需要一次顺序IO读取所有的数据，而无须任何的随机IO，问题在于查询的列特别多的时候，会占用大量的存储空间，无法存储大量的数据

- 总结：

  当需要排序的列的总大小超过max_length_for_sort_data定义的字节，mysql会选择双次排序，反之使用单次排序，当然，用户可以设置此参数的值来选择排序的方式

▼ 优化特定类型的查询

▼ 优化count()查询

count()是特殊的函数，有两种不同的作用，一种是某个列值的数量，也可以统计行数

1. count(1)、count(id)、count(*) 效率基本一样——通过执行计划和执行之间对别测试的，已经验证过。
2. 总有人认为myisam的count函数比较快，这是有前提条件的，只有没有任何where条件的count(*)才是比较快的
3. 使用近似值

   在某些应用场景中，不需要完全精确的值，可以参考使用近似值来代替，比如可以使用explain来获取近似的值

   其实在很多OLAP的应用中，需要计算某一个列值的基数，有一个计算近似值的算法叫hyperloglog。
4. 更复杂的优化

   一般情况下，count()需要扫描大量的行才能获取精确的数据，其实很难优化，在实际操作的时候可以考虑使用索引覆盖扫描，或者增加汇总表，或者增加外部缓存系统。

▼ 优化关联查询

- 确保on或者using子句中的列上有索引，在创建索引的时候就要考虑到关联的顺序

  当表A和表B使用列C关联的时候，如果优化器的关联顺序是B、A，那么就不需要再B表的对应列上建上索引，没有用到的索引只会带来额外的负担，一般情况下来说，只需要在关联顺序中的第二个表的相应列上创建索引

- 确保任何的groupby和order by中的表达式只涉及到一个表中的列，这样mysql才有可能使用索引来优化这个过程

▼ 优化子查询

子查询的优化最重要的优化建议是尽可能使用关联查询代替

子查询会暂时放在临时表、临时表也是IO

▼ 优化limit分页

在很多应用场景中我们需要将数据进行分页，一般会使用limit加上偏移量的方法实现，同时加上合适的 orderby 的子句，如果这种方式有索引的帮助，效率通常不错，否则的化需要进行大量的文件排序操作，还有一种情况，当偏移量非常大的时候，前面的大部分数据都会被抛弃，这样的代价太高。

要优化这种查询的话，要么是在页面中限制分页的数量，要么优化大偏移量的性能

- 优化此类查询的最简单的办法就是尽可能地使用覆盖索引，而不是查询所有的列

  select film_id,description from film order by title limit 50,5

  explain select film.film_id,film.description from film inner join (select film_id from film order by title limit 50,5) as lim using(film_id);

  查看执行计划扫描的行数

▼ 优化union查询

mysql总是通过创建并填充临时表的方式来执行union查询，因此很多优化策略在union查询中都没法很好的使用。经常需要手工的将where、limit、order by等子句下推到各个子查询中，以便优化器可以充分利用这些条件进行优化

- 除非确实需要服务器消除重复的行，否则一定要使用union all，因此没有all关键字，mysql会在查询的时候给临时表加上distinct的关键字，这个操作的代价很高

▼ 推荐使用用户自定义变量

用户自定义变量是一个容易被遗忘的mysql特性，但是如果能够用好，在某些场景下可以写出非常高效的查询语句，在查询中混合使用过程化和关系话逻辑的时候，自定义变量会非常有用。

用户自定义变量是一个用来存储内容的临时容器，在连接mysql的整个过程中都存在。

- 自定义变量的使用

  set @one :=1

  set @min_actor :=(select min(actor_id) from actor)

  set @last_week :=current_date-interval 1 week;

- 自定义变量的限制

  1、无法使用查询缓存

  2、不能在使用常量或者标识符的地方使用自定义变量，例如表名、列名或者limit子句

  3、用户自定义变量的生命周期是在一个连接中有效，所以不能用它们来做连接间的通信

  4、不能显式地声明自定义变量地类型

  5、mysql优化器在某些场景下可能会将这些变量优化掉，这可能导致代码不按预想地方式运行

  6、赋值符号：=的优先级非常低，所以在使用赋值表达式的时候应该明确的使用括号

7、使用未定义变量不会产生任何语法错误

- 自定义变量的使用案例

1. 优化排名语句

a、在给一个变量赋值的同时使用这个变量

```
select actor_id,@rownum:=@rownum+1 as rownum from actor limit 10;
```

b、查询获取演过最多电影的前10名演员，然后根据出演电影次数做一个排名

```
SET @actor_number:=0;
SELECT actor_id,cnt,@actor_number:=@actor_number +1 FROM (SELECT
actor_id,COUNT(1) AS cnt FROM film_actor GROUP BY actor_id ORDER BY cnt DESC
LIMIT 10) t;
```

2. 避免重新查询刚刚更新的数据

当需要高效的更新一条记录的时间戳，同时希望查询当前记录中存放的时间戳是什么

```
update t1 set lastUpdated=now() where id =1;
select lastUpdated from t1 where id =1;


update t1 set lastupdated = now() where id = 1 and @now:=now();
select @now;
```

3. 确定取值的顺序

在赋值和读取变量的时候可能是在查询的不同阶段

```
set @rownum:=0;
select actor_id,@rownum:=@rownum+1 as cnt from actor where @rownum<=1;
因为where和select在查询的不同阶段执行，所以看到查询到两条记录，这不符合预期


解决这个问题的关键在于让变量的赋值和取值发生在执行查询的同一阶段：
set @rownum:=0;
select actor_id,@rownum as cnt from actor where (@rownum:=@rownum+1)<=1;
```