

8 种很坑的 SQL 错误用法

程序员追风 Java后端 2019-11-21

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 程序员追风

链接 | juejin.im/post/5dd15451e51d453b3d3d4329

上篇 | 一千万 MySQL 学习笔记

1、LIMIT 语句

分页查询是最常用的场景之一, 但也通常也是最容易出问题的地方。比如对于下面简单的语句, 一般 DBA 想到的办法是在 type, name, create_time 字段上加组合索引。这样条件排序都能有效的利用到索引, 性能迅速提升。

```
SELECT *
FROM   operation
WHERE  type = 'SQLStats'
       AND name = 'SlowLog'
ORDER  BY create_time
LIMIT  1000, 10;
```

好吧, 可能90%以上的 DBA 解决该问题就到此为止。但当 LIMIT 子句变成 “LIMIT 1000000,10” 时, 程序员仍然会抱怨: 我只取10条记录为什么还是慢?

要知道数据库也并不知道第1000000条记录从什么地方开始, 即使有索引也需要从头计算一次。出现这种性能问题, 多数情形下是程序员偷懒了。

在前端数据浏览翻页, 或者大数据分批导出等场景下, 是可以将上一页的最大值当成参数作为查询条件的。SQL 重新设计如下:

```
SELECT  *
FROM    operation
WHERE   type = 'SQLStats'
AND     name = 'SlowLog'
AND     create_time > '2017-03-16 14:00:00'
ORDER  BY create_time limit 10;
```

在新设计下查询时间基本固定, 不会随着数据量的增长而发生变化。

2、隐式转换

SQL语句中查询变量和字段定义类型不匹配是另一个常见的错误。比如下面的语句:

```

mysql> explain extended SELECT *
    > FROM my_balance b
    > WHERE b.bpn = 14000000123
    >       AND b.isverified IS NULL ;
mysql> show warnings;
| Warning | 1739 | Cannot use ref access on index 'bpn' due
      to type or collation conversion on field 'bpn'

```

其中字段 bpn 的定义为 varchar(20), MySQL 的策略是将字符串转换为数字之后再比较。函数作用于表字段，索引失效。

上述情况可能是应用程序框架自动填入的参数，而不是程序员的原意。现在应用框架很多很繁杂，使用方便的同时也小心它可能给自己挖坑。

3、关联更新、删除

虽然 MySQL5.6 引入了物化特性，但需要特别注意它目前仅仅针对查询语句的优化。对于更新或删除需要手工重写成 JOIN。

比如下面 UPDATE 语句，MySQL 实际执行的是循环/嵌套子查询 (DEPENDENT SUBQUERY)，其执行时间可想而知。

```

UPDATE operation o
SET status = 'applying'
WHERE o.id IN (SELECT id
               FROM (SELECT o.id,
                           o.status
                  FROM operation o
                 WHERE o.group = 123
                   AND o.status NOT IN ( 'done' )
                  ORDER BY o.parent,
                           o.id
                  LIMIT 1) t);

```

执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	o	index		PRIMARY	8		24	Using where; Using temporary
2	DEPENDENT SUBQUERY								Impossible WHERE noticed after reading const tables
3	DERIVED	o	ref	idx_2, idx_5	idx_5	8	const	1	Using where; Using filesort

重写为 JOIN 之后，子查询的选择模式从 DEPENDENT SUBQUERY 变成 DERIVED，执行速度大大加快，从7秒降低到2毫秒

```

UPDATE operation o
JOIN (SELECT o.id,
            o.status
           FROM operation o
          WHERE o.group = 123
            AND o.status NOT IN ( 'done' )
           ORDER BY o.parent,
                    o.id
           LIMIT 1) t
      ON o.id = t.id
SET status = 'applying'

```

执行计划简化为：

id select_type table type possible_keys key key_len ref rows Extra
1 PRIMARY Impossible WHERE noticed after reading const tables
2 DERIVED o ref idx_2, idx_5 idx_5 8 const 1 Using where; Using filesort

4、混合排序

MySQL 不能利用索引进行混合排序。但在某些场景，还是有机会使用特殊方法提升性能的。

```
SELECT *
FROM my_order o
    INNER JOIN my_appraise a ON a.orderid = o.id
ORDER BY a.is_reply ASC,
         a.appraise_time DESC
LIMIT 0, 20
```

执行计划显示为全表扫描：

id select_type table type possible_keys key key_len ref rows Extra
1 SIMPLE a ALL idx_orderid NULL NULL NULL 1967647 Using filesort
1 SIMPLE o eq_ref PRIMARY PRIMARY 122 a.orderid 1 NULL

由于 is_reply 只有0和1两种状态，我们按照下面的方法重写后，执行时间从1.58秒降低到2毫秒。

```
SELECT *
FROM ((SELECT *
       FROM my_order o
       INNER JOIN my_appraise a
              ON a.orderid = o.id
              AND is_reply = 0
       ORDER BY appraise_time DESC
       LIMIT 0, 20)
UNION ALL
(SELECT *
       FROM my_order o
       INNER JOIN my_appraise a
              ON a.orderid = o.id
              AND is_reply = 1
       ORDER BY appraise_time DESC
       LIMIT 0, 20)) t
ORDER BY is_reply ASC,
         appraisetime DESC
LIMIT 20;
```

5、EXISTS语句

MySQL 对待 EXISTS 子句时，仍然采用嵌套子查询的执行方式。如下面的 SQL 语句：

```

SELECT *
FROM my_neighbor n
    LEFT JOIN my_neighbor_apply sra
        ON n.id = sra.neighbor_id
        AND sra.user_id = 'xxx'
WHERE n.topic_status < 4
    AND EXISTS(SELECT 1
                FROM message_info m
                WHERE n.id = m.neighbor_id
                AND m.inuser = 'xxx')
    AND n.topic_type <> 5

```

执行计划为：

id select_type	table type	possible_keys	key	key_len	ref	rows	Extra
1 PRIMARY	n	ALL	NULL	NULL	NULL	1086041	Using where
1 PRIMARY	sra	ref	idx_user_id	123	const	1	Using where
2 DEPENDENT SUBQUERY	m	ref	idx_message_info	122	const	1	Using index condition; Using where

去掉 exists 改为 join，能够避免嵌套子查询，将执行时间从1.93秒降低为1毫秒。

```

SELECT *
FROM my_neighbor n
INNER JOIN message_info m
    ON n.id = m.neighbor_id
    AND m.inuser = 'xxx'
LEFT JOIN my_neighbor_apply sra
    ON n.id = sra.neighbor_id
    AND sra.user_id = 'xxx'
WHERE n.topic_status < 4
    AND n.topic_type <> 5

```

新的执行计划：

id select_type table type	possible_keys	key	key_len	ref	rows	Extra
1 SIMPLE	m	ref	idx_message_info	122	const	1 Using index condition
1 SIMPLE	n	eq_ref	PRIMARY	122	neighbor_id	1 Using where
1 SIMPLE	sra	ref	idx_user_id	123	const	1 Using where

6、条件推

外部查询条件不能够下推到复杂的视图或子查询的情况有：

- 聚合子查询；
- 含有 LIMIT 的子查询；
- UNION 或 UNION ALL 子查询；
- 输出字段中的子查询；

如下面的语句，从执行计划可以看出其条件作用于聚合子查询之后

```
SELECT *
FROM   (SELECT target,
              Count(*)
       FROM   operation
       GROUP  BY target) t
WHERE  target = 'rm-xxxx'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	PRIMARY		<derived2>	ref	<auto_key0>	<auto_key0>	514	const	2	Using where
2	DERIVED		operation	index	idx_4	idx_4	519	NULL	20	Using index

确定从语义上查询条件可以直接下推后，重写如下：

```
SELECT target,
       Count(*)
  FROM   operation
 WHERE  target = 'rm-xxxx'
 GROUP  BY target
```

执行计划变为：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	operation	ref	idx_4	idx_4	514	const	1	Using where; Using index	

7、提前缩小范围

先上初始 SQL 语句：

```
SELECT *
  FROM my_order o
    LEFT JOIN my_userinfo u
      ON o.uid = u.uid
    LEFT JOIN my_productinfo p
      ON o.pid = p.pid
 WHERE ( o.display = 0 )
   AND ( o.ostatus = 1 )
 ORDER BY o.selltime DESC
LIMIT 0, 15
```

数为90万，时间消耗为12秒。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	o	ALL	NULL	NULL	NULL	NULL	909119	Using where; Using temporary; Using filesort
1	SIMPLE	u	eq_ref	PRIMARY	PRIMARY	4	o.uid	1	NULL
1	SIMPLE	p	ALL	PRIMARY	NULL	NULL	NULL	6	Using where; Using join buffer (Block Nested Loop)

由于最后 WHERE 条件以及排序均针对最左主表，因此可以先对 my_order 排序提前缩小数据量再做左连接。SQL 重写后如下，执行时间缩小为1毫秒左右。

```
SELECT *
FROM (
SELECT *
FROM my_order o
WHERE ( o.display = 0 )
      AND ( o.ostaus = 1 )
ORDER BY o.selltime DESC
LIMIT 0, 15
) o
LEFT JOIN my_userinfo u
      ON o.uid = u.uid
LEFT JOIN my_productinfo p
      ON o.pid = p.pid
ORDER BY o.selltime DESC
limit 0, 15
```

再检查执行计划：子查询物化后 (select_type=DERIVED) 参与 JOIN。虽然估算行扫描仍然为90万，但是利用了索引以及 LIMIT 子句后，实际执行时间变得很小。

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	15	Using temporary; Using filesort
1	PRIMARY	u	eq_ref	PRIMARY	PRIMARY	4	o.uid	1	NULL
1	PRIMARY	p	ALL	PRIMARY	NULL	NULL	NULL	6	Using where; Using join buffer (Block Nested Loop)
2	DERIVED	o	index	NULL	idx_1	5	NULL	909112	Using where

8、中间结果集下推

再来看下面这个已经初步优化过的例子(左连接中的主表优先作用查询条件)：

```

SELECT      a.*,
            c.allocated
FROM        (
            SELECT      resourceid
            FROM        my_distribute d
            WHERE       isdelete = 0
            AND        cusmanagercode = '1234567'
            ORDER BY   salecode limit 20) a
LEFT JOIN
(
            SELECT      resourcesid,  sum(ifnull(allocation, 0) * 12345) allocated
            FROM        my_resources
            GROUP BY   resourcesid) c
ON          a.resourceid = c.resourcesid

```

那么该语句还存在其它问题吗？不难看出子查询 c 是全表聚合查询，在表数量特别大的情况下会导致整个语句的性能下降。

其实对于子查询 c，左连接最后结果集只关心能和主表 resourceid 能匹配的数据。因此我们可以重写语句如下，执行时间从原来的2秒下降到2毫秒。

```

SELECT      a.*,
            c.allocated
FROM        (
            SELECT      resourceid
            FROM        my_distribute d
            WHERE       isdelete = 0
            AND        cusmanagercode = '1234567'
            ORDER BY   salecode limit 20) a
LEFT JOIN
(
            SELECT      resourcesid,  sum(ifnull(allocation, 0) * 12345) allocated
            FROM        my_resources r,
            (
                    SELECT      resourceid
                    FROM        my_distribute d
                    WHERE       isdelete = 0
                    AND        cusmanagercode = '1234567'
                    ORDER BY   salecode limit 20) a
            WHERE       r.resourcesid = a.resourcesid
            GROUP BY   resourcesid) c
ON          a.resourceid = c.resourcesid

```

但是子查询 a 在我们的SQL语句中出现了多次。这种写法不仅存在额外的开销，还使得整个语句显的繁杂。使用 WITH 语句再次重写：

```
WITH a AS
(
    SELECT      resourceid
    FROM        my_distribute d
    WHERE       isdelete = 0
    AND         cusmanagercode = '1234567'
    ORDER BY    salecode limit 20)
SELECT      a.*,
            c.allocated
FROM        a
LEFT JOIN   (
    SELECT      resourcesid, sum(ifnull(allocation, 0) * 12345) allocated
    FROM        my_resources r,
                a
    WHERE       r.resourcesid = a.resourcesid
    GROUP BY   resourcesid) c
ON          a.resourceid = c.resourcesid
```

总结

数据库编译器产生执行计划，决定着SQL的实际执行方式。但是编译器只是尽力服务，所有数据库的编译器都不是尽善尽美的。

上述提到的多数场景，在其它数据库中也存在性能问题。了解数据库编译器的特性，才能避规其短处，写出高性能的SQL语句。

程序员在设计数据模型以及编写SQL语句时，要把算法的思想或意识带进来。

编写复杂SQL语句要养成使用 WITH 语句的习惯。简洁且思路清晰的SQL语句也能减小数据库的负担。

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. [一千行 MySQL 学习笔记](#)
2. [一份工作坚持多久跳槽最合适?](#)
3. [项目中常用到的 19 条 MySQL 优化](#)
4. [零基础认识 Spring Boot](#)
5. [团队开发中 Git 最佳实践](#)



声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

MySQL 如何查找删除重复行？

神父，我有罪 Java后端 2019-10-27

点击上方 Java后端, 选择 [设为星标](#)

优质文章，及时送达

作者 | 神父、我有罪

来源 | www.xaprb.com

如何查找重复行

第一步是定义什么样的行才是重复行。多数情况下很简单：它们某一列具有相同的值。本文采用这一定义，或许你对“重复”的定义比这复杂，你需要对sql做些修改。本文要用到的数据样本：

```
create table test(id int not null primary key, day date not null);
```

```
insert into test(id, day) values(1, '2006-10-08');
insert into test(id, day) values(2, '2006-10-08');
insert into test(id, day) values(3, '2006-10-09');
```

```
select * from test;
```

id	day
1	2006-10-08
2	2006-10-08
3	2006-10-09

前面两行在day字段具有相同的值，因此如何我将他们当做重复行，这里有一查询语句可以查找。查询语句使用GROUP BY子句把具有相同字段值的行归为一组，然后计算组的大小。

```
select day, count(*) from test GROUP BY day;
```

day	count(*)
2006-10-08	2
2006-10-09	1

重复行的组大小大于1。如何希望只显示重复行，必须使用HAVING子句，比如

```
select day, count(*) from test group by day HAVING count(*) > 1;
```

day	count(*)
2006-10-08	2

这是基本的技巧：根据具有相同值的字段分组，然后知显示大小大于1的组。

为什么不能使用WHERE子句？因为WHERE子句过滤的是分组之前的行，HAVING子句过滤的是分组之后的行。

如何删除重复行

一个相关的问题是如何删除重复行。一个常见的任务是，重复行只保留一行，其他删除，然后你可以创建适当的索引，防止以后再有重复的行写入数据库。

同样，首先是弄清楚重复行的定义。你要保留的是哪一行呢？第一行，或者某个字段具有最大值的行？本文中，假设要保留的是第一行——id字段具有最小值的行，意味着你要删除其他的行。

也许最简单的方法是通过临时表。尤其对于MYSQL，有些限制是不能在一个查询语句中select的同时update一个表。简单起见，这里只用到了临时表的方法。

我们的任务是：删除所有重复行，除了分组中id字段具有最小值的行。因此，需要找出大小大于1的分组，以及希望保留的行。你可以使用MIN()函数。这里的语句是创建临时表，以及查找需要用DELETE删除的行。

```
create temporary table to_delete (day date not null, min_id int not null);

insert into to_delete(day, min_id)
    select day, MIN(id) from test group by day having count(*) > 1;

select * from to_delete;
+-----+-----+
| day | min_id |
+-----+-----+
| 2006-10-08 | 1 |
+-----+-----+
```

有了这些数据，你可以开始删除“脏数据”行了。可以有几种方法，各有优劣（详见我的文章many-to-one problems in SQL），但这里不做详细比较，只是说明在支持查询子句的关系数据库中，使用的标准方法。

tips:欢迎关注微信公众号：Java后端，获取每日技术博文推送。

```
delete from test
where exists(
    select * from to_delete
    where to_delete.day = test.day and to_delete.min_id <> test.id
)
```

如何查找多列上的重复行

有人最近问到这样的问题：我的一个表上有两个字段b和c，分别关联到其他两个表的b和c字段。我想要找出在b字段或者c字段上具有重复值的行。

乍看很难明白，通过对话后我理解了：他想要对b和c分别创建unique索引。如上所述，查找在某一字段上具有重复值的行很简单，只要用group分组，然后计算组的大小。并且查找全部字段重复的行也很简单，只要把所有字段放到group子句。但如果是判断b字段重复或者c字段重复，问题困难得多。这里提问者用到的样本数据

```
create table a_b_c(
    a int not null primary key auto_increment,
    b int,
    c int
);
```

```
insert into a_b_c(b,c) values (1,1);
insert into a_b_c(b,c) values (1,2);
insert into a_b_c(b,c) values (1,3);
insert into a_b_c(b,c) values (2,1);
insert into a_b_c(b,c) values (2,2);
insert into a_b_c(b,c) values (2,3);
insert into a_b_c(b,c) values (3,1);
insert into a_b_c(b,c) values (3,2);
insert into a_b_c(b,c) values (3,3);
```

现在，你可以轻易看到表里面有一些重复的行，但找不到两行具有相同的二元组{b, c}。这就是为什么问题会变得困难了。

错误的查询语句

如果把两列放在一起分组，你会得到不同的结果，具体看如何分组和计算大小。提问者恰恰是困在了这里。有时候查询语句找到一些重复行却漏了其他的。这是他用到了查询

```
select b, c, count(*) from a_b_c
group by b, c
having count(distinct b > 1)
or count(distinct c > 1);
```

结果返回所有的行，因为COUNT(*)总是1.为什么？因为 >1 写在COUNT()里面。这个错误很容易被忽略，事实上等效于

```
select b, c, count(*) from a_b_c
group by b, c
having count(1)
or count(1);
```

为什么？因为(b > 1)是一个布尔值，根本不是你想要的结果。你要的是

```
select b, c, count(*) from a_b_c
group by b, c
having count(distinct b) > 1
or count(distinct c) > 1;
```

返回空结果。很显然，因为没有重复的{b,c}。这人试了很多其他的OR和AND的组合，用来分组的是一个字段，计算大小的是另一个字段，像这样

```
select b, count(*) from a_b_c group by b having count(distinct c) > 1;
+-----+-----+
| b | count(*) |
+-----+-----+
| 1 | 3 |
| 2 | 3 |
| 3 | 3 |
+-----+-----+
```

没有一个能够找出全部的重复行。而且最令人沮丧的是，对于某些情况，这种语句是有效的，如果错误地以为就是这么写法，然而对于另外的情况，很可能得到错误结果。

事实上，单纯用GROUP BY 是不可行的。为什么？因为当你对某一字段使用group by时，就会把另一字段的值分散到不同的分组里。对这些字段排序可以看到这些效果，正如分组做的那样。首先，对b字段排序，看看它是如何分组的

a	b	c
7	1	1
8	1	2
9	1	3
10	2	1
11	2	2
12	2	3
13	3	1
14	3	2
15	3	3

当你对b字段排序（分组），相同值的c被分到不同的组，因此不能用COUNT(DISTINCT c)来计算大小。COUNT()之类的内部函数只作用于同一个分组，对于不同分组的行就无能为力了。类似，如果排序的是c字段，相同值的b也会分到不同的组，无论如何是不能达到我们的目的的。

几种正确的方法

也许最简单的方法是分别对某个字段查找重复行，然后用UNION拼在一起，像这样：

```
select b as value, count(*) as cnt, 'b' as what_col
from a_b_c group by b having count(*) > 1
union
select c as value, count(*) as cnt, 'c' as what_col
from a_b_c group by c having count(*) > 1;
+-----+-----+
| value | cnt | what_col |
+-----+-----+
| 1 | 3 | b |
| 2 | 3 | b |
| 3 | 3 | b |
| 1 | 3 | c |
| 2 | 3 | c |
| 3 | 3 | c |
+-----+-----+
```

输出what_col字段为了提示重复的是哪个字段。另一个办法是使用嵌套查询：

```
select a, b, c from a_b_c
where b in (select b from a_b_c group by b having count(*) > 1)
  or c in (select c from a_b_c group by c having count(*) > 1);
+---+---+
| a | b | c |
+---+---+
| 7 | 1 | 1 |
| 8 | 1 | 2 |
| 9 | 1 | 3 |
| 10 | 2 | 1 |
| 11 | 2 | 2 |
| 12 | 2 | 3 |
| 13 | 3 | 1 |
| 14 | 3 | 2 |
| 15 | 3 | 3 |
+---+---+
```

这种方法的效率要比使用UNION低许多，并且显示每一重复的行，而不是重复的字段值。还有一种方法，将自己跟group的嵌套查询结果联表查询。写法比较复杂，但对于复杂的数据或者对效率有较高要求的情况，是很有必要的。

```
select a,a_b_c.b,a_b_c.c
from a_b_c
left outer join (
    select b from a_b_c group by b having count(*) > 1
) as b on a_b_c.b = b.b
left outer join (
    select c from a_b_c group by c having count(*) > 1
) as c on a_b_c.c = c.c
where b.b is not null or c.c is not null
```

以上方法可行，我敢肯定还有其他的方法。如果UNION能用，我想会是最简单不过的了。

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. [10 个让你笑的合不拢嘴的 GitHub 项目](#)
2. [当我遵循了这 16 条规范写代码](#)
3. [理解 IntelliJ IDEA 的项目配置和 Web 部署](#)
4. [Java 开发中常用的 4 种加密方法](#)
5. [团队开发中 Git 最佳实践](#)



声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

↑ 点击蓝字 关注 Java 后端

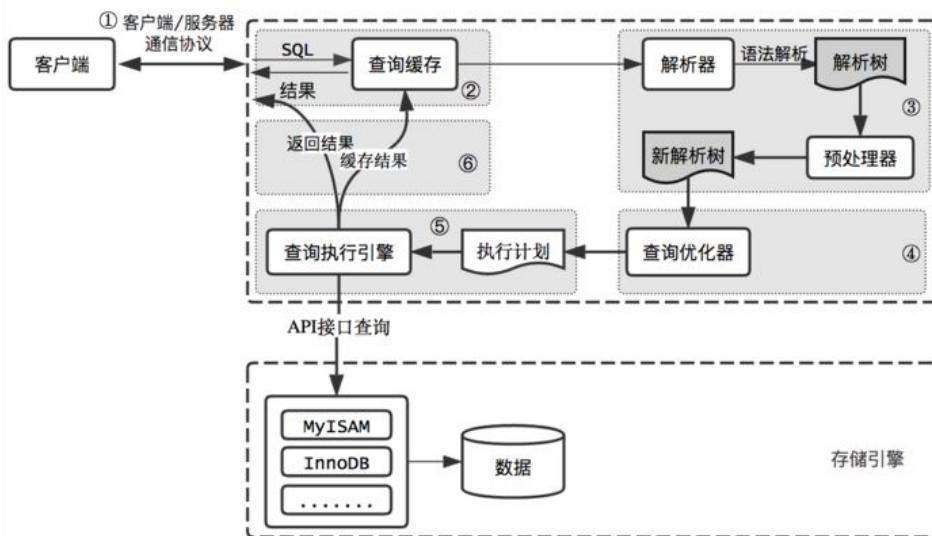
作者 | 惨绿少年

链接 | <https://clsn.io/clsn/lx287.html>

一、前言

MySQL调优对于很多程序员而言，都是一个非常棘手的问题，多数情况都是因为对数据库出现问题的情况和处理思路不清晰。在进行MySQL的优化之前必须要了解的就是MySQL的查询过程，很多的查询优化工作实际上就是遵循一些原则让MySQL的优化器能够按照预想的合理方式运行而已。

今天给大家讲解MySQL的优化实战，助你高薪之路顺畅！



二、优化的哲学

注意：优化有风险，涉足需谨慎！

1、优化可能带来的问题

- 1) 优化不总是对一个单纯的环境进行，还很可能是一个复杂的已投产的系统。
- 2) 优化手段本来就有很大的风险，只不过你没能力意识到和预见到！
- 3) 任何的技术可以解决一个问题，但必然存在带来一个问题的风险！
- 4) 对于优化来说解决问题而带来的问题，控制在可接受的范围内才是有成果。
- 5) 保持现状或出现更差的情况都是失败！

2、优化的需求

- 1) 稳定性和业务可持续性,通常比性能更重要!
- 2) 优化不可避免涉及到变更, 变更就有风险!
- 3) 优化使性能变好, 维持和变差是等概率事件!
- 4) 切记优化,应该是各部门协同, 共同参与的工作, 任何单一部门都不能对数据库进行优化!
- 5) 所以优化工作,是由业务需要驱使的!!!

3、优化由谁参与

在进行数据库优化时, 应由数据库管理员、业务部门代表、应用程序架构师、应用程序设计人员、应用程序开发人员、硬件及系统管理员、存储管理员等, 业务相关人员共同参与。

tips: 大家可以关注微信公众号: Java后端, 获取更多优秀博文推送。

三、优化思路

1、优化什么

在数据库优化上有两个主要方面: 即安全与性能。

- 1) 安全 --> 数据可持续性
- 2) 性能 --> 数据的高性能访问

2、优化的范围有哪些

存储、主机和操作系统方面:

- 1) 主机架构稳定性
- 2) I/O规划及配置
- 3) Swap交换分区
- 4) OS内核参数和网络问题

应用程序方面:

- 1) 应用程序稳定性
- 2) SQL语句性能
- 3) 串行访问资源
- 4) 性能欠佳会话管理
- 5) 这个应用适不适合用MySQL

数据库优化方面:

- 1) 内存
- 2) 数据库结构(物理&逻辑)
- 3) 实例配置

说明: 不管是在, 设计系统, 定位问题还是优化, 都可以按照这个顺序执行。

3、优化维度

数据库优化维度有四个：

硬件、系统配置、数据库表结构、SQL及索引。

优化选择：

1) 优化成本: 硬件>系统配置>数据库表结构>SQL及索引

2) 优化效果: 硬件<系统配置<数据库表结构<SQL及索引

四、优化工具有啥？

1、数据库层面

检查问题常用工具：

1 mysql	
2 mysqladmin	mysql客户端，可进行管理操作
3 mysqlshow	功能强大的查看shell命令
4 show [SESSION GLOBAL] variables	查看数据库参数信
5 息	
6 SHOW [SESSION GLOBAL] STATUS	查看数据库的状态信
7 息	
8 information_schema	获取元数据的方法
9 SHOW ENGINE INNODB STATUS	Innodb引擎的所有状
10 态	
11 SHOW PROCESSLIST	查看当前所有连接session状
12 态	
explain	获取查询语句的执行计划
show index	查看表的索引信息
slow_log	记录慢查询语句
mysqldumpslow	分析slowlog文件的

不常用但好用的工具：

1 zabbix	监控主机、系统、数据库（部署zabbix监控平台）
2)	
3 pt-query-digest	分析慢日志
4 志	
5 mysqlslap	分析慢日志
6 志	
7 sysbench	压力测试工具
具	
mysql profiling	统计数据库整体状态工具
Performance Schema	mysql性能状态统计的数据
workbench	管理、备份、监控、分析、优化工具（比较费资源）
)	

2、数据库层面问题解决思路

一般应急调优的思路：

针对突然的业务办理卡顿，无法进行正常的业务处理！需要立马解决的场景！

```
1 1、show processlist
2
3 2、explain select id ,name from stu where name='clsn'; # ALL  id name age  sex
4
5         select id,name from stu  where id=2-1 函数 结果集>30
6 ;
7
8     show index from table;
9
10 3、通过执行计划判断，索引问题（有没有、合不合理）或者语句本身问题
11
12 4、show status like '%Lock%';    # 查询锁状
13 态

kill SESSION_ID;    # 杀掉有问题的session
```

常规调优思路：

针对业务周期性的卡顿，例如在每天10-11点业务特别慢，但是还能够使用，过了这段时间就好了。

- 1) 查看slowlog，分析slowlog，分析出查询慢的语句。
- 2) 按照一定优先级，进行一个一个的排查所有慢语句。
- 3) 分析top sql，进行explain调试，查看语句执行时间。
- 4) 调整索引或语句本身。

3、系统层面

cpu方面：

vmstat、sar top、htop、nmon、mpstat

内存：

free、ps -aux、

IO设备（磁盘、网络）：

iostat、ss、netstat、iptraf、iftop、lsof、

vmstat 命令说明：

Procs: r显示有多少进程正在等待CPU时间。b显示处于不可中断的休眠的进程数量。在等待I/O
Memory: swpd显示被交换到磁盘的数据块的数量。未被使用的数据块，用户缓冲数据块，用于操作系统的数据块的数量
Swap: 操作系统每秒从磁盘上交换到内存和从内存交换到磁盘的数据块的数量。s1和s0最好是0
lo: 每秒从设备中读入b1的写入到设备b0的数据块的数量。反映了磁盘I/O
System: 显示了每秒发生中断的数量(in)和上下文交换(cs)的数量
Cpu: 显示用于运行用户代码，系统代码，空闲，等待I/O的CPU时间

iostat命令说明

实例命令：iostat -dk 1 5

iostat -d -k -x 5 （查看设备使用率（%util）和响应时间（await））

- 1) tps：该设备每秒的传输次数。“一次传输”意思是“一次I/O请求”。多个逻辑请求可能会被合并为“一次I/O请求”。
- 2) iops：硬件出厂的时候，厂家定义的一个每秒最大的IO次数，“一次传输”请求的大小是未知的。

- 3) kBread/s: 每秒从设备 (drive expressed) 读取的数据量;
- 4) KBwrtn/s: 每秒向设备 (drive expressed) 写入的数据量;
- 5) kBread: 读取的总数据量; 7、kBwrtn: 写入的总数量数据量; 这些单位都为Kilobytes。

4. 系统层面问题解决办法

你认为到底负载高好，还是低好呢？

在实际的生产中，一般认为cpu只要不超过90%都没什么问题。

当然不排除下面这些特殊情况：

问题一：cpu负载高，IO负载低

1、内存不够 2、磁盘性能差 3、SQL问题 ----->去数据库层，进一步排查sql问题 4、IO出问题了（磁盘到临界了、raid设计不好、raid降级、锁、在单位时间内tps过高） 5、tps过高：大量的小数据IO、大量的全表扫描

问题二：IO负载高，cpu负载低

1、大量小的IO写操作：2、autocommit，产生大量小IO 3、IO/PS,磁盘的一个定值，硬件出厂的时候，厂家定义的一个每秒最大的IO次数。4、大量大的IO写操作 5、SQL问题的几率比较大

问题三：IO和cpu负载都很高

硬件不够了或sql存在问题

五、基础优化

1. 优化思路

定位问题点：

硬件 --> 系统 --> 应用 --> 数据库 --> 架构（高可用、读写分离、分库分表）

处理方向：

明确优化目标、性能和安全的折中、防患未然

2. 硬件优化

主机方面：

- 1) 根据数据库类型，主机CPU选择、内存容量选择、磁盘选择
- 2) 平衡内存和磁盘资源
- 3) 随机的I/O和顺序的I/O
- 4) 主机 RAID卡的BBU(Battery Backup Unit)关闭

cpu的选择：

- 1) cpu的两个关键因素：核数、主频
- 2) 根据不同的业务类型进行选择
- 3) cpu密集型：计算比较多，OLTP 主频很高的cpu、核数还要多
- 4) IO密集型：查询比较，OLAP 核数要多，主频不一定高的

内存的选择：

1) OLAP类型数据库，需要更多内存，和数据获取量级有关。

2) OLTP类型数据一般内存是cpu核心数量的2倍到4倍，没有最佳实践。

存储方面：

1) 根据存储数据种类的不同，选择不同的存储设备

2) 配置合理的RAID级别(raid5、 raid10、 热备盘)

3) 对与操作系统来讲，不需要太特殊的选择，最好做好冗余（raid1）（ssd、 sas、 sata）

raid卡：主机raid卡选择：

1) 实现操作系统磁盘的冗余（raid1）

2) 平衡内存和磁盘资源

3) 随机的I/O和顺序的I/O

4) 主机 RAID卡的BBU(Battery Backup Unit)要关闭。

网络设备方面：

使用流量支持更高的网络设备（交换机、路由器、网线、网卡、HBA卡）

注意：以上这些规划应该在初始设计系统时就应该考虑好。

3、服务器硬件优化

1) 物理状态灯：

2) 自带管理设备：远程控制卡（FENCE设备：ipmi ilo idarc），开关机、硬件监控。

3) 第三方的监控软件、设备（snmp、 agent）对物理设施进行监控

4) 存储设备：自带的监控平台。EMC2（hp收购了），日立（hds），IBM低端OEM hds，高端存储是自己技术，华为存储

4、系统优化

Cpu：

基本不需要调整，在硬件选择方面下功夫即可。

内存：

基本不需要调整，在硬件选择方面下功夫即可。

SWAP：

MySQL尽量避免使用swap。阿里云的服务器中默认swap为0

IO：

1) raid、no lvm、ext4或xfs、ssd、IO调度策略

2) Swap调整(不使用swap分区)

```
1 /proc/sys/vm/swappiness的内容改成0 (临时)，/etc/sysctl.conf上添加vm.swappiness=0 (永久)
```

这个参数决定了Linux是倾向于使用swap，还是倾向于释放文件系统cache。在内存紧张的情况下，数值越低越倾向于释放文件系

统cache。当然，这个参数只能减少使用swap的概率，并不能避免Linux使用swap。修改MySQL的配置参数

innodbflushmethod，开启O_DIRECT模式。这种情况下，InnoDB的buffer pool会直接绕过文件系统cache来访问磁盘，但是

redo log依旧会使用文件系统cache。值得注意的是，Redo log是覆写模式的，即使使用了文件系统的cache，也不会占用太多。

IO调度策略：

```
1 vi /boot/grub/grub.conf
2 更改到如下内容：
3 kernel /boot/vmlinuz-2.6.18-8.el5 ro root=LABEL=/ elevator=deadline rhgb quiet
```

5、系统参数调整

Linux系统内核参数优化：

```
1 vim /etc/sysctl.conf
2     net.ipv4.ip_local_port_range = 1024 65535    # 用户端口范围
3     net.ipv4.tcp_max_syn_backlog = 4096
4
5     net.ipv4.tcp_fin_timeout = 30
6
7     fs.file-max=65535          # 系统最大文件句柄，控制的是能打开文件最大数量
```

用户限制参数（mysql可以不设置以下配置）：

```
1 vim /etc/security/limits.conf
2 * soft nproc 6553
3 5
4 * hard nproc 6553
5
6 * soft nofile 6553
7
8 * hard nofile 6553
9
```

6、应用优化

业务应用和数据库应用独立,防火墙：iptables、selinux等其他无用服务(关闭)：

```
1 chkconfig --level 23456 acpid off
2 f
3 chkconfig --level 23456 anacron off
4 f
5 chkconfig --level 23456 autofs off
6 f
7 chkconfig --level 23456 avahi-daemon off
8 f
9 chkconfig --level 23456 bluetooth off
10 chkconfig --level 23456 cups off
11 chkconfig --level 23456 firstboot off
12 chkconfig --level 23456 haldaemon off
13 chkconfig --level 23456 hplip off
14 f
15 chkconfig --level 23456 isstablos off
```

```
15 chkconfig --level 23456 iptables off
chkconfig --level 23456 isdn off
chkconfig --level 23456 pcscd on
f
chkconfig --level 23456 sendmail on
f
chkconfig --level 23456 yum-updatesd on
f
```

安装图形界面的服务器不要启动图形界面 runlevel 3,另外，思考将来我们的业务是否真的需要MySQL，还是使用其他种类的数据库。用数据库的最高境界就是不用数据库。

六、数据库优化

SQL优化方向：

执行计划、索引、SQL改写

架构优化方向：

高可用架构、高性能架构、分库分表

1、数据库参数优化

调整：

实例整体（高级优化，扩展）

```
1 thread_concurrency          # 并发线程数量个数
2 sort_buffer_size            # 排序缓存
3 存
4 read_buffer_size            # 顺序读取缓存
5 存
6 read_rnd_buffer_size        # 随机读取缓存
7 存
key_buffer_size               # 索引缓存
8 存
thread_cache_size             # (1G->8, 2G->16, 3G->32, >3G->64)
```

连接层（基础优化）

设置合理的连接客户和连接方式

```
1 max_connections              # 最大连接数，看交易笔数设置
2
3 max_connect_errors           # 最大错误连接数，能大则大
4
5 connect_timeout               # 连接超时
6
7 max_user_connections          # 最大用户连接数
8
skip-name-resolve                # 跳过域名解析
```

```
wait_timeout          # 等待超时  
back_log             # 可以在堆栈中的连接数量
```

SQL层（基础优化）

querycachesize：查询缓存-->>>OLAP类型数据库,需要重点加大此内存缓存.

- 1) 但是一般不会超过GB.
- 2) 对于经常被修改的数据，缓存会立马失效。
- 3) 我们可以实用内存数据库（redis、memecache），替代他的功能。

2、存储引擎层（innodb基础优化参数）

```
1 default-storage-engine  
2 innodb_buffer_pool_size      # 没有固定大小, 50%测试值, 看看情况再微调。但是尽量设置不要超过物理内存70  
3 %  
4 innodb_file_per_table=(1,0)  
5 innodb_flush_log_at_trx_commit=(0,1,2) # 1是最安全的, 0是性能最高, 2折中  
6 binlog_sync  
7 Innodb_flush_method=(0_DIRECT, fdatasync)  
8 innodb_log_buffer_size       # 100M以  
9 下  
10 innodb_log_file_size        # 100M 以  
11 下  
12 innodb_log_files_in_group   # 5个成员以下, 一般2-3个够用 (iblogfile0-N)  
13 innodb_max_dirty_pages_pct  # 达到百分之75的时候刷写 内存脏页到磁盘。  
14 log_bin  
    max_binlog_cache_size       # 可以不设  
    置  
    max_binlog_size            # 可以不设  
    置  
    innodb_additional_mem_pool_size # 小于2G内存的机器, 推荐值是20M。32G内存以上100M
```

tips：大家可以关注微信公众号：Java后端，获取更多优秀博文推送。

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. 感受 Lambda 之美，推荐收藏，需要时查阅
2. 如何优雅的导出 Excel
3. 文艺互联网公司 vs 二逼互联网公司
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践



喜欢文章，点个在看

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

MySQL 用 limit 为什么会影响性能?

zhangyachen Java后端 2019-11-30

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | zhangyachen

编辑 | Java之间

来源 | zhangyachen.github.io

一, 前言

首先说明一下MySQL的版本:

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.7.17    |
+-----+
1 row in set (0.00 sec)
```

表结构:

```
mysql> desc test;
+-----+-----+-----+-----+-----+
| Field | Type            | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+
| id    | bigint(20) unsigned | NO   | PRI | NULL    | auto_increment |
| val   | int(10) unsigned    | NO   | MUL | 0       |                |
| source | int(10) unsigned    | NO   |      | 0       |                |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

id为自增主键, val为非唯一索引。

灌入大量数据, 共500万:

```
mysql> select count(*) from test;
+-----+
| count(*) |
+-----+
| 5242882 |
+-----+
1 row in set (4.25 sec)
```

我们知道, 当limit offset rows中的offset很大时, 会出现效率问题:

```
mysql> select * from test where val=4 limit 300000,5;
+-----+-----+-----+
| id   | val  | source |
+-----+-----+-----+
| 3327622 |    4 |      4 |
| 3327632 |    4 |      4 |
| 3327642 |    4 |      4 |
| 3327652 |    4 |      4 |
| 3327662 |    4 |      4 |
+-----+-----+-----+
5 rows in set (15.98 sec)
```

为了达到相同的目的，我们一般会改写成如下语句：

```
mysql> select * from test a inner join (select id from test where val=4 limit 300000,5) b on a.id=b.id;
+-----+-----+-----+-----+
| id   | val  | source | id   |
+-----+-----+-----+-----+
| 3327622 |    4 |      4 | 3327622 |
| 3327632 |    4 |      4 | 3327632 |
| 3327642 |    4 |      4 | 3327642 |
| 3327652 |    4 |      4 | 3327652 |
| 3327662 |    4 |      4 | 3327662 |
+-----+-----+-----+-----+
5 rows in set (0.38 sec)
```

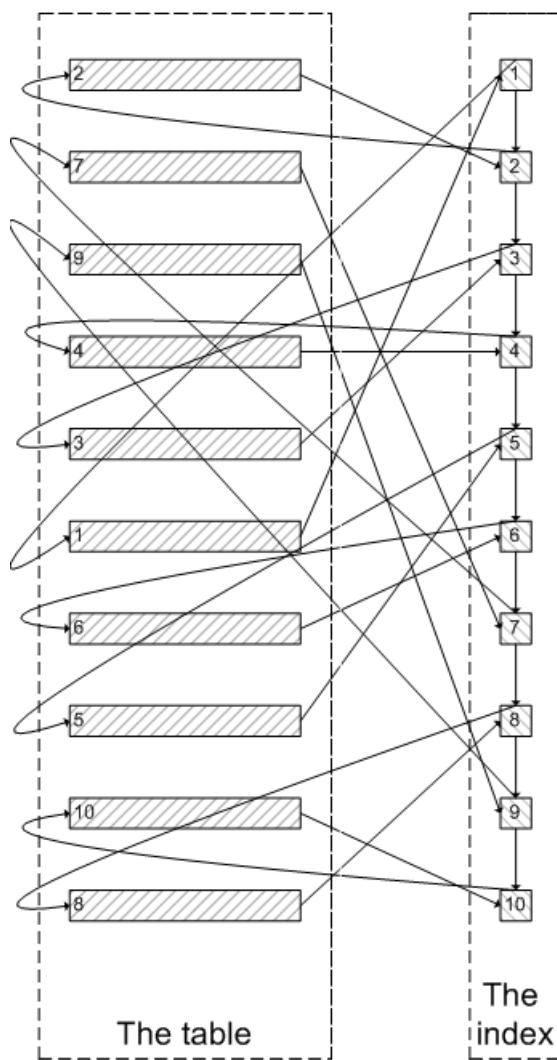
时间相差很明显。

为什么会出现上面的结果？我们看一下`select * from test where val=4 limit 300000,5;`的查询过程：

查询到索引叶子节点数据。

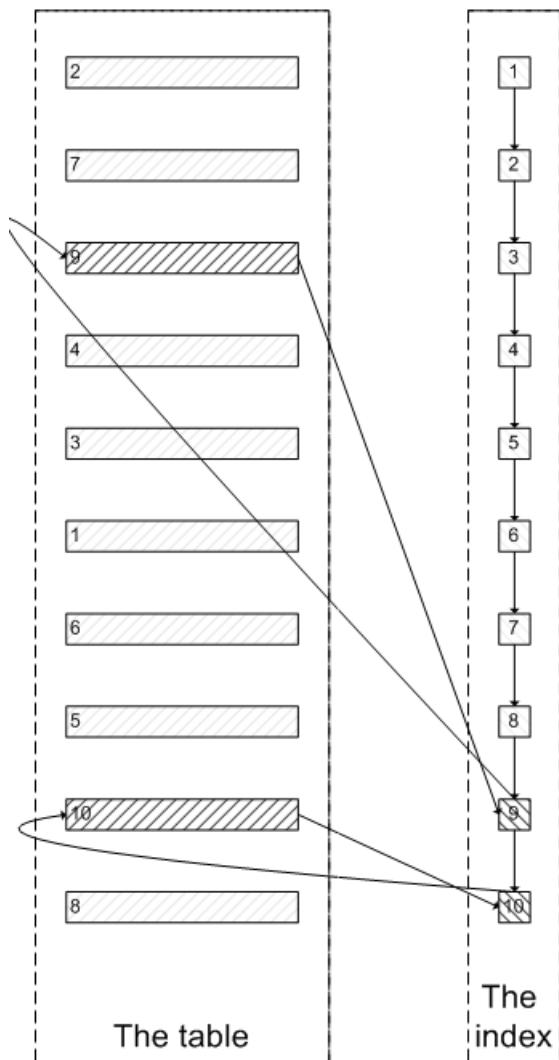
根据叶子节点上的主键值去聚簇索引上查询需要的全部字段值。

类似于下面这张图：



像上面这样，需要查询300005次索引节点，查询300005次聚簇索引的数据，最后再将结果过滤掉前300000条，取出最后5条。MySQL耗费了大量随机I/O在查询聚簇索引的数据上，而有300000次随机I/O查询到的数据是不会出现在结果集当中的。

肯定会有人问：既然一开始是利用索引的，为什么不先沿着索引叶子节点查询到最后需要的5个节点，然后再去聚簇索引中查询实际数据。这样只需要5次随机I/O，类似于下面图片的过程：



其实我也想问这个问题。

证实

下面我们实际操作一下来证实上述的推论：

为了证实 `select * from test where val=4 limit 300000,5` 是扫描300005个索引节点和300005个聚簇索引上的数据节点，我们需要知道MySQL有没有办法统计在一个sql中通过索引节点查询数据节点的次数。我先试了Handler_read_*系列，很遗憾没有一个变量能满足条件。

我只能通过间接的方式来证实：

InnoDB中有buffer pool。里面存有最近访问过的数据页，包括数据页和索引页。所以我们需要运行两个sql，来比较buffer pool中的数据页的数量。预测结果是运行 `select * from test a inner join (select id from test where val=4 limit 300000,5) b>`之后，buffer pool中的数据页的数量远远少于 `select * from test where val=4 limit 300000,5;` 对应的数量，因为前一个sql只访问5次数据页，而后一个sql访问300005次数据页。

```
select * from test where val=4 limit 300000,5
```

```
mysql> select index_name, count(*) from information_schema.INNODB_BUFFER_PAGE where INDEX_NAME in('val', 'primary') and TABLE_NAME like '%test%' group by index_name;
Empty set (0.04 sec)
```

可以看出，目前buffer pool中没有关于test表的数据页。

```
mysql> select * from test where val=4 limit 300000,5;
+-----+-----+-----+
| id | val | source |
+-----+-----+-----+
| 3327622 | 4 | 4 |
| 3327632 | 4 | 4 |
| 3327642 | 4 | 4 |
| 3327652 | 4 | 4 |
| 3327662 | 4 | 4 |
+-----+-----+-----+
5 rows in set (26.19 sec)
```

```
mysql> select index_name,count(*) from information_schema.INNODB_BUFFER_PAGE where INDEX_NAME in('val','primary') and TABLE_NAME like '%test%' group by index_name;
+-----+-----+
| index_name | count(*) |
+-----+-----+
| PRIMARY | 4098 |
| val | 208 |
+-----+-----+
2 rows in set (0.04 sec)
```

可以看出，此时buffer pool中关于test表有4098个数据页，208个索引页。

```
select * from test a inner join (select id from test where val=4 limit 300000,5) b>为了防止上次试验的影响，我们需要清空buffer pool，重启mysql。
```

```
mysqladmin shutdown
/usr/local/bin/mysqld_safe &
```

```
mysql> select index_name,count(*) from information_schema.INNODB_BUFFER_PAGE where INDEX_NAME in('val','primary') and TABLE_NAME like '%test%' group by index_name;
Empty set (0.03 sec)
```

运行sql:

```
mysql> select * from test a inner join (select id from test where val=4 limit 300000,5) b on a.id=b.id;
+-----+-----+-----+-----+
| id | val | source | id |
+-----+-----+-----+-----+
| 3327622 | 4 | 4 | 3327622 |
| 3327632 | 4 | 4 | 3327632 |
| 3327642 | 4 | 4 | 3327642 |
| 3327652 | 4 | 4 | 3327652 |
| 3327662 | 4 | 4 | 3327662 |
+-----+-----+-----+-----+
5 rows in set (0.09 sec)

mysql> select index_name,count(*) from information_schema.INNODB_BUFFER_PAGE where INDEX_NAME in('val','primary') and TABLE_NAME like '%test%' group by index_name;
+-----+-----+
| index_name | count(*) |
+-----+-----+
| PRIMARY | 5 |
| val | 390 |
+-----+-----+
2 rows in set (0.03 sec)
```

我们可以看明显的看出两者的差别：第一个sql加载了4098个数据页到buffer pool，而第二个sql只加载了5个数据页到buffer pool。符合我们的预测。也证实了为什么第一个sql会慢：读取大量的无用数据行（300000），最后却抛弃掉。

而且这会造成一个问题：加载了很多热点不是很高的数据页到buffer pool，会造成buffer pool的污染，占用buffer pool的空间。

遇到的问题

为了在每次重启时确保清空buffer pool，我们需要关闭innodb_buffer_pool_dump_at_shutdown和innodb_buffer_pool_load_at_startup，这两个选项能够控制数据库关闭时dump出buffer pool中的数据和在数据库开启时载入在磁盘上备份buffer pool的数据。

参考资料：

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. [如果我是面试官，我会问你 Spring 那些问题？](#)
2. [Spring Boot 整合 Spring-cache](#)
3. [我们再来聊一聊 Java 的单例吧](#)
4. [我采访了一位 Pornhub 工程师](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章，点个在看

MySQL 用得好好的，为啥非要转 ES？

张sir Java后端 2019-11-16

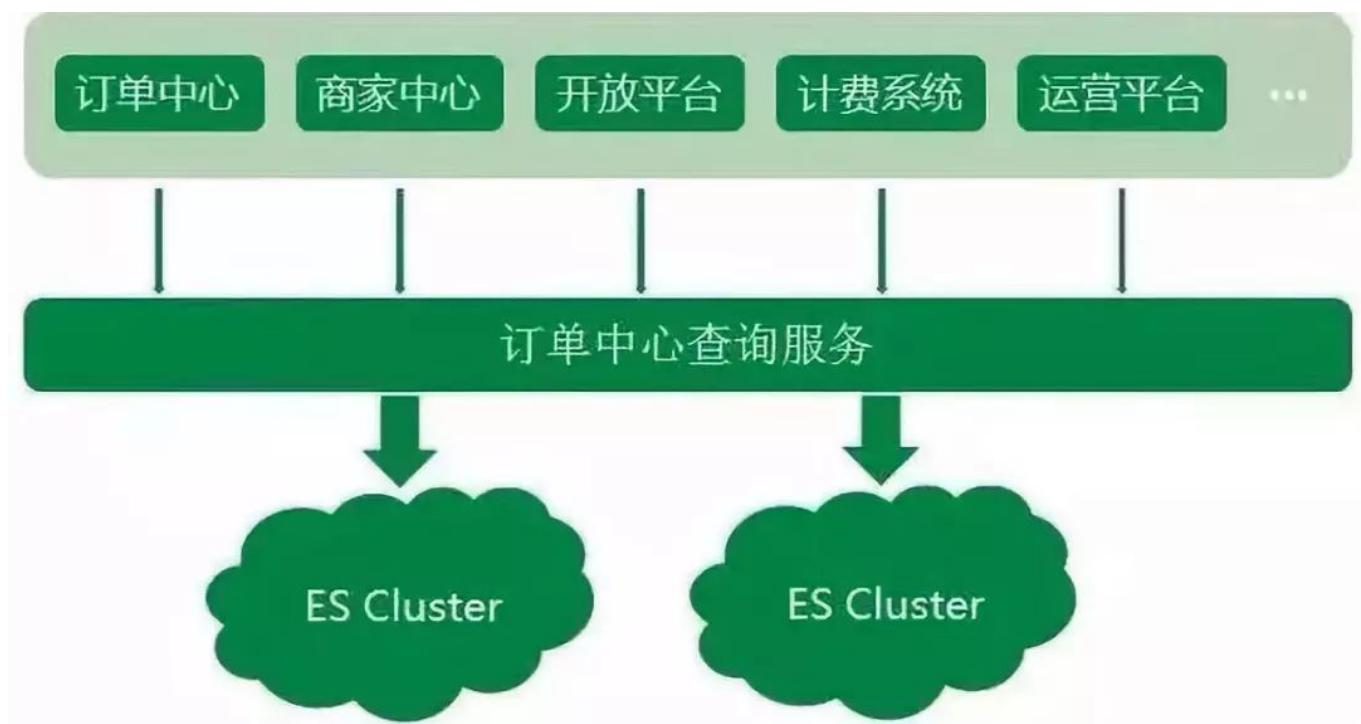
点击上方 Java后端, 选择 [设为星标](#)

优质文章，及时送达

来源：京东技术

京东到家订单中心系统业务中，无论是外部商家的订单生产，或是内部上下游系统的依赖，订单查询的调用量都非常大，造成了订单数据读多写少的情况。

我们把订单数据存储在MySQL中，但显然只通过DB来支撑大量的查询是不可取的。同时对于一些复杂的查询，MySQL支持得不够友好，所以订单中心系统使用了Elasticsearch来承载订单查询的主要压力。



Elasticsearch作为一款功能强大的分布式搜索引擎，支持近实时的存储、搜索数据，在京东到家订单系统中发挥着巨大作用，目前订单中心ES集群存储数据量达到10亿个文档，日均查询量达到5亿。

随着京东到家近几年业务的快速发展，订单中心ES架设方案也不断演进，发展至今ES集群架设是一套实时互备方案，很好地保障了ES集群读写的稳定性，下面就给大家介绍一下这个历程以及过程中遇到的一些坑。

ES 集群架构演进之路

1、初始阶段

订单中心ES初始阶段如一张白纸，架设方案基本没有，很多配置都是保持集群默认配置。整个集群部署在集团的弹性云上，ES集群的节点以及机器部署都比较混乱。同时按照集群维度来看，一个ES集群会有单点问题，显然对于订单中心业务来说也是不被允许的。

2、集群隔离阶段

和很多业务一样，ES集群采用的混布的方式。但由于订单中心ES存储的是线上订单数据，偶尔会发生混布集群抢占系统大量资源，导致整个订单中心ES服务异常。

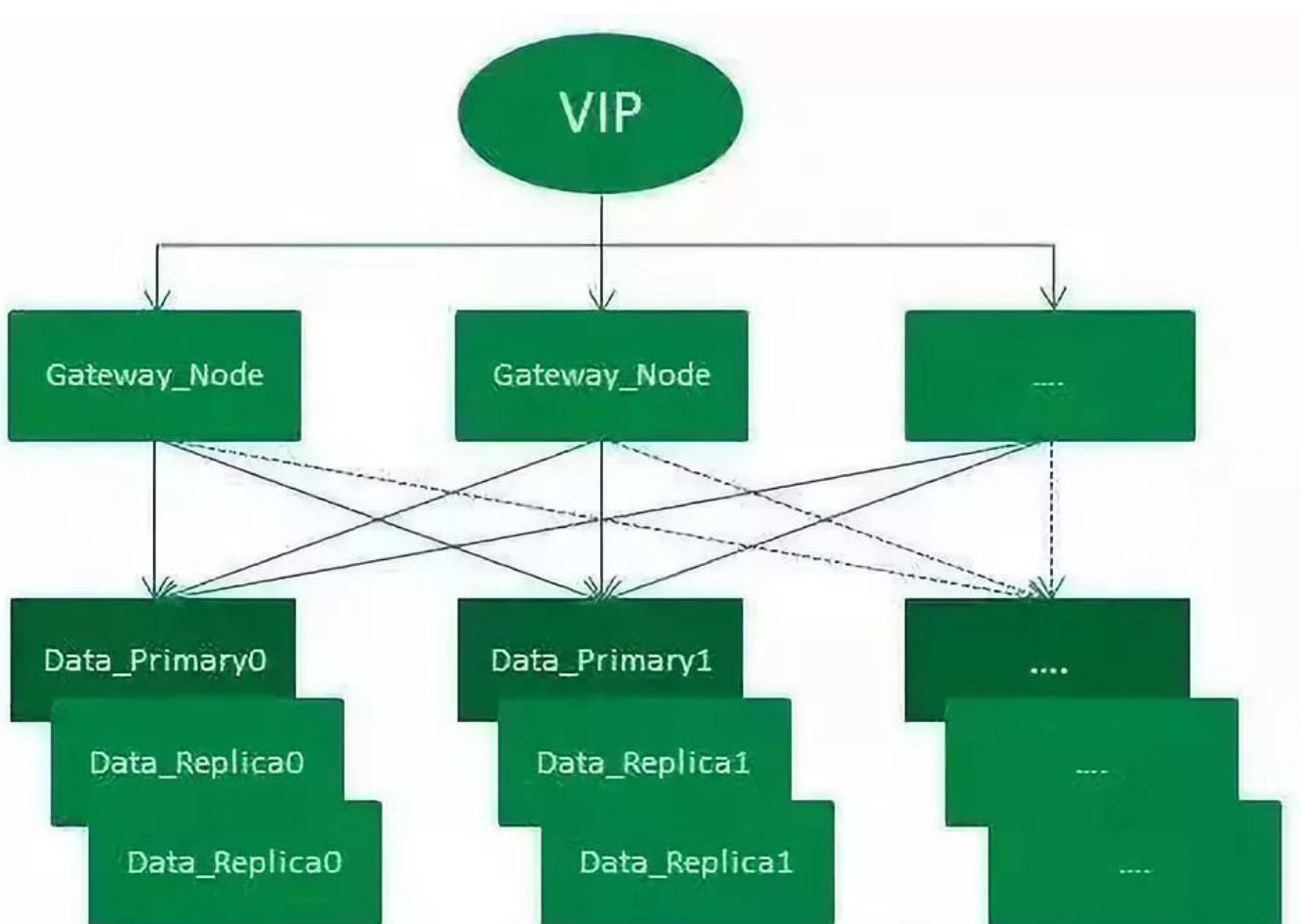
显然任何影响到订单查询稳定性的情况都是无法容忍的，所以针对于这个情况，先是针对订单中心ES所在的弹性云，迁出那些系统资源抢占很高的集群节点，ES集群状况稍有好转。但随着集群数据不断增加，弹性云配置已经不太能满足ES集群，且为了完全的物理隔离，最终干脆将订单中心ES集群部署到高配置的物理机上，ES集群性能又得到提升。

3、节点副本调优阶段

ES的性能跟硬件资源有很大关系，当ES集群单独部署到物理机器上时，集群内部的节点并不是独占整台物理机资源，在集群运行的时候同一物理机上的节点仍会出现资源抢占的问题。所以在这种情况下，为了让ES单个节点能够使用最大程度的机器资源，采用每个ES节点部署在单独一台物理机上方式。

但紧接着，问题又来了，如果单个节点出现瓶颈了呢？我们应该怎么再优化呢？

ES查询的原理，当请求打到某号分片的时候，如果没有指定分片类型（Preference参数）查询，请求会负载到对应分片号的各个节点上。而集群默认副本配置是一主一副，针对此情况，我们想到了扩容副本的方式，由默认的一主一副变为一主二副，同时增加相应物理机。

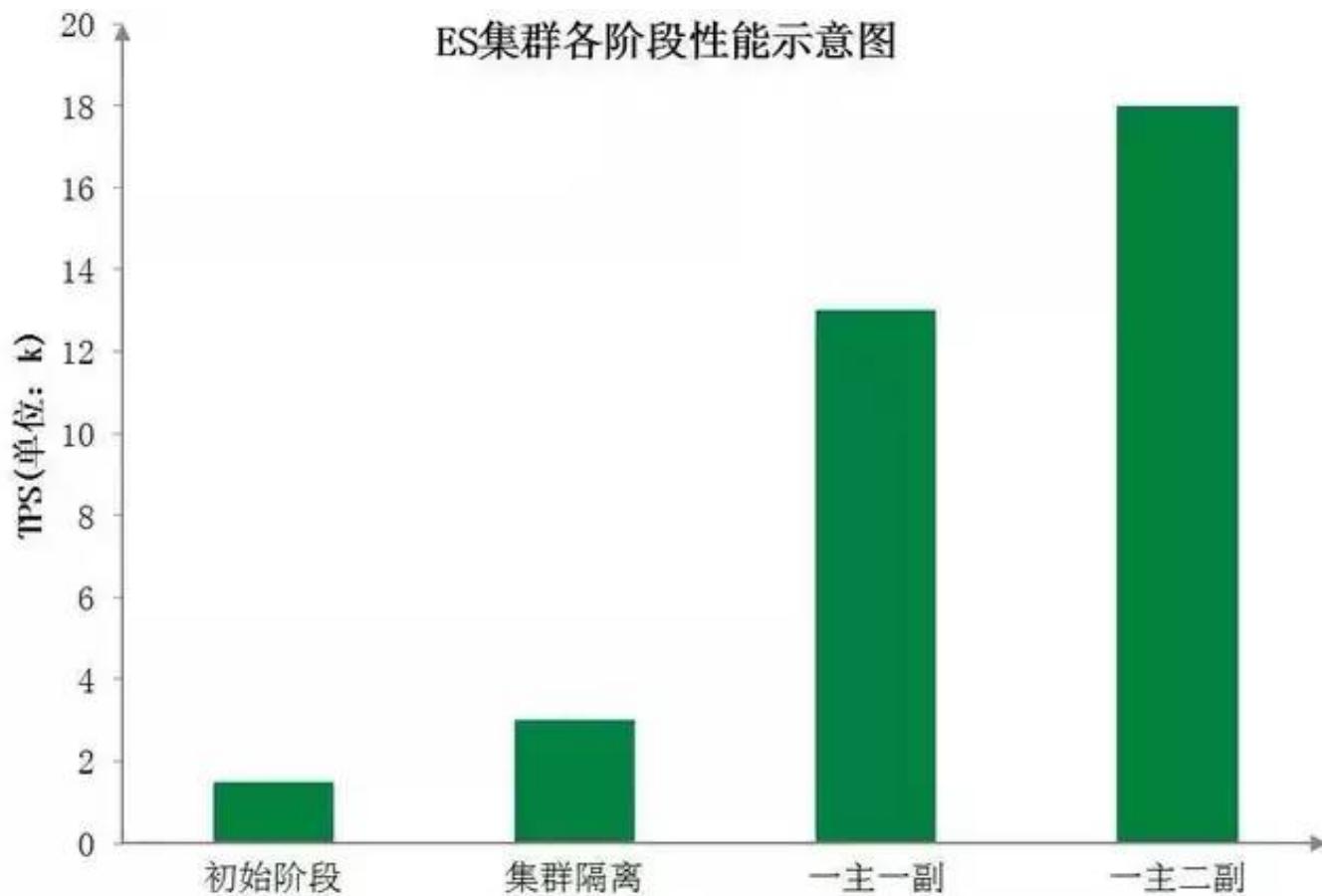


订单中心ES集群架设示意图

如图，整个架设方式通过VIP来负载均衡外部请求：

整个集群有一套主分片，二套副本分片（一主二副），从网关节点转发过来的请求，会在打到数据节点之前通过轮询的方式进行均衡。集群增加一套副本并扩容机器的方式，增加了集群吞吐量，从而提升了整个集群查询性能。

下图为订单中心ES集群各阶段性能示意图，直观地展示了各阶段优化后ES集群性能的显著提升：



当然分片数量和分片副本数量并不是越多越好，在此阶段，我们对选择适当的分片数量做了进一步探索。分片数可以理解为MySQL中的分库分表，而当前订单中心ES查询主要分为两类：单ID查询以及分页查询。

分片数越大，集群横向扩容规模也更大，根据分片路由的单ID查询吞吐量也能大大提升，但聚合的分页查询性能则将降低；分片数越小，集群横向扩容规模也更小，单ID的查询性能也会下降，但分页查询的性能将会提升。

所以如何均衡分片数量和现有查询业务，我们做了很多次调整压测，最终选择了集群性能较好的分片数。

4、主从集群调整阶段

到此，订单中心的ES集群已经初具规模，但由于订单中心业务时效性要求高，对ES查询稳定性要求也高，如果集群中有节点发生异常，查询服务会受到影响，从而影响到整个订单生产流程。很明显这种异常情况是致命的，所以为了应对这种情况，我们初步设想是增加一个备用集群，当主集群发生异常时，可以实时的将查询流量降级到备用集群。

那备用集群应该怎么来搭？主备之间数据如何同步？备用集群应该存储什么样的数据？

考虑到ES集群暂时没有很好的主备方案，同时为了更好地控制ES数据写入，我们采用业务双写的方式来搭设主备集群。每次业务操作需要写入ES数据时，同步写入主集群数据，然后异步写入备集群数据。同时由于大部分ES查询的流量都来源于近几天的订

单，且订单中心数据库数据已有一套归档机制，将指定天数之前已经关闭的订单转移到历史订单库。

所以归档机制中增加删除备集群文档的逻辑，让新搭建的备集群存储的订单数据与订单中心线上数据库中的数据量保持一致。同时使用ZK在查询服务中做了流量控制开关，保证查询流量能够实时降级到备集群。在此，订单中心主从集群完成，ES查询服务稳定性大大提升。



5、现今：实时互备双集群阶段

期间由于主集群ES版本是较低的1.7，而现今ES稳定版本都已经迭代到6.x，新版本的ES不仅性能方面优化很大，更提供了一些新的好用的功能，所以我们对主集群进行了一次版本升级，直接从原来的1.7升级到6.x版本。

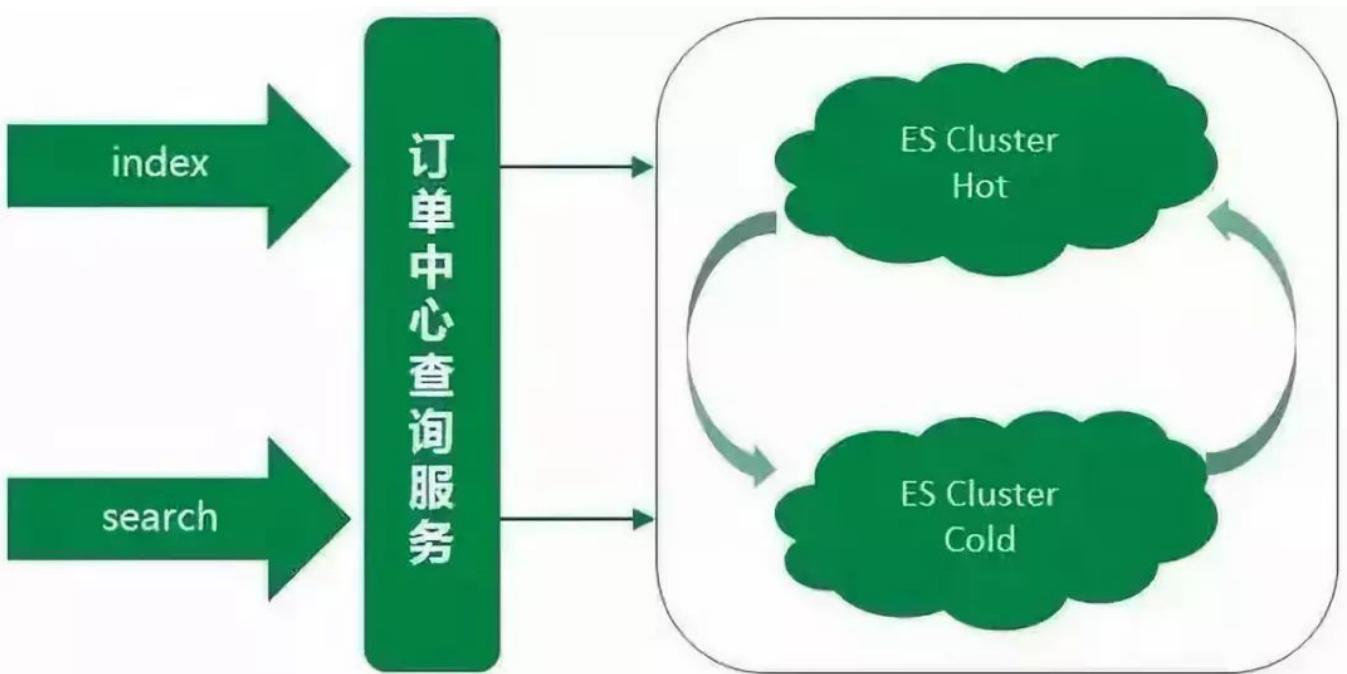
集群升级的过程繁琐而漫长，不但需要保证线上业务无任何影响，平滑无感知升级，同时由于ES集群暂不支持从1.7到6.x跨越多个版本的数据迁移，所以需要通过重建索引的方式来升级主集群，具体升级过程就不在此赘述了。

主集群升级的时候必不可免地会发生不可用的情况，但对于订单中心ES查询服务，这种情况是不允许的。所以在升级的阶段中，备集群暂时顶上充当主集群，来支撑所有的线上ES查询，保证升级过程不影响正常线上服务。同时针对线上业务，我们对两个集群做了重新的规划定义，承担的线上查询流量也做了重新的划分。

备集群存储的是线上近几天的热点数据，数据规模远小于主集群，大约是主集群文档数的十分之一。集群数据量小，在相同的集群部署规模下，备集群的性能要优于主集群。

然而在线上真实场景中，线上大部分查询流量也来源于热点数据，所以用备集群来承载这些热点数据的查询，而备集群也慢慢演变成一个热数据集群。之前的主集群存储的是全量数据，用该集群来支撑剩余较小部分的查询流量，这部分查询主要是需要搜索全量订单的特殊场景查询以及订单中心系统内部查询等，而主集群也慢慢演变成一个冷数据集群。

同时备集群增加一键降级到主集群的功能，两个集群地位同等重要，但都可以各自降级到另一个集群。双写策略也优化为：假设有AB集群，正常同步方式写主（A集群）异步方式写备（B集群）。A集群发生异常时，同步写B集群（主），异步写A集群（备）。



ES 订单数据的同步方案

MySQL数据同步到ES中，大致总结可以分为两种方案：

- 方案1：监听MySQL的Binlog，分析Binlog将数据同步到ES集群中。
- 方案2：直接通过ES API将数据写入到ES集群中。

考虑到订单系统ES服务的业务特殊性，对于订单数据的实时性较高，显然监听Binlog的方式相当于异步同步，有可能会产生较大的延时性。且方案1实质上跟方案2类似，但又引入了新的系统，维护成本也增高。所以订单中心ES采用了直接通过ES API写入订单数据的方式，该方式简洁灵活，能够很好的满足订单中心数据同步到ES的需求。

由于ES订单数据的同步采用的是在业务中写入的方式，当新建或更新文档发生异常时，如果重试势必会影响业务正常操作的响应时间。

所以每次业务操作只更新一次ES，如果发生错误或者异常，在数据库中插入一条补救任务，有Worker任务会实时地扫这些数据，以数据库订单数据为基准来再次更新ES数据。通过此种补偿机制，来保证ES数据与数据库订单数据的最终一致性。

遇到的一些坑

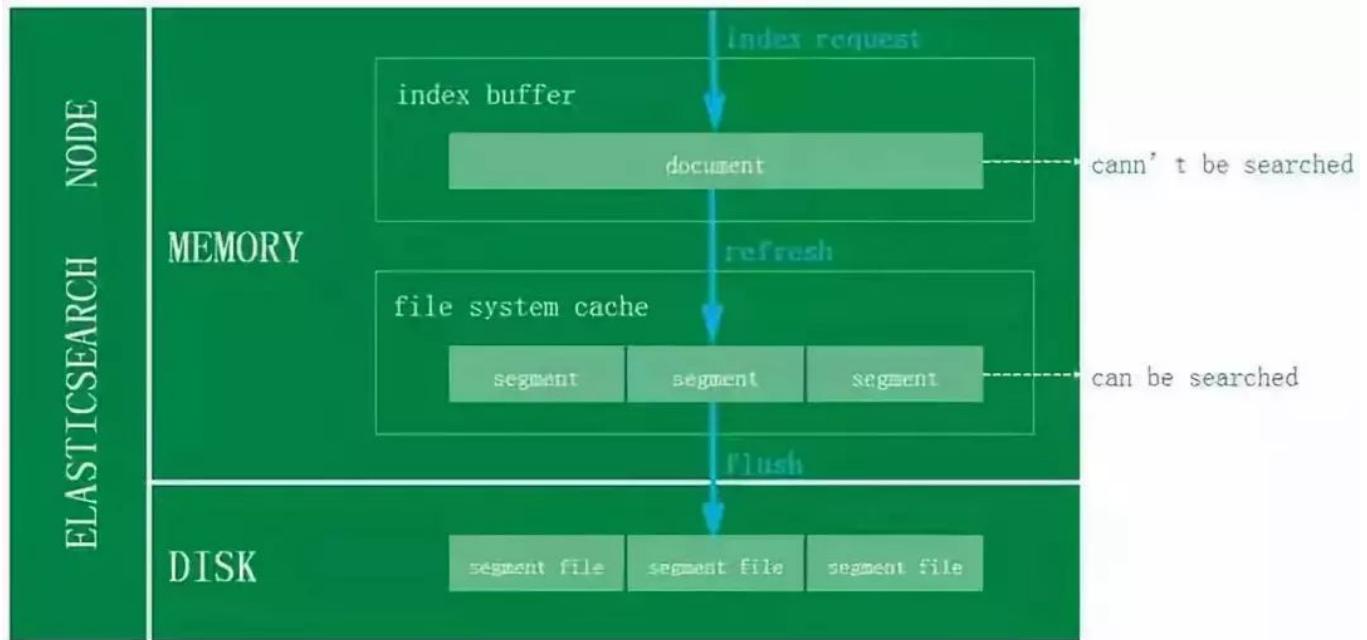
1、实时性要求高的查询走DB

对于ES写入机制的有了解的同学可能会知道，新增的文档会被收集到Indexing Buffer，然后写入到文件系统缓存中，到了文件系统缓存中就可以像其他的文件一样被索引到。

然而默认情况文档从Indexing Buffer到文件系统缓存（即Refresh操作）是每秒分片自动刷新，所以这就是我们说ES是近实时搜索而非实时的原因：文档的变化并不是立即对搜索可见，但会在一秒之内变为可见。

当前订单系统ES采用的是默认Refresh配置，故对于那些订单数据实时性比较高的业务，直接走数据库查询，保证数据的准确

性。



2、避免深分页查询

ES集群的分页查询支持from和size参数，查询的时候，每个分片必须构造一个长度为from+size的优先队列，然后回传到网关节点，网关节点再对这些优先队列进行排序找到正确的size个文档。

假设在一个有6个主分片的索引中，from为10000，size为10，每个分片必须产生10010个结果，在网关节点中汇聚合并60060个结果，最终找到符合要求的10个文档。

由此可见，当from足够大的时候，就算不发生OOM，也会影响到CPU和带宽等，从而影响到整个集群的性能。所以应该避免深分页查询，尽量不去使用。

3、FieldData与Doc Values

FieldData

线上查询出现偶尔超时的情况，通过调试查询语句，定位到是跟排序有关系。排序在es1.x版本使用的是FieldData结构，FieldData占用的是JVM Heap内存，JVM内存是有限，对于FieldData Cache会设定一个阈值。

如果空间不足时，使用最久未使用（LRU）算法移除FieldData，同时加载新的FieldData Cache，加载的过程需要消耗系统资源，且耗时很大。所以导致这个查询的响应时间暴涨，甚至影响整个集群的性能。针对这种问题，解决方式是采用Doc Values。

Doc Values

Doc Values是一种列式的数据存储结构，跟FieldData很类似，但其存储位置是在Lucene文件中，即不会占用JVM Heap。随着ES版本的迭代，Doc Values比FieldData更加稳定，Doc Values在2.x起为默认设置。

总结

架构的快速迭代源于业务的快速发展，正是由于近几年到家业务的高速发展，订单中心的架构也不断优化升级。而架构方案没有最好的，只有最合适的，相信再过几年，订单中心的架构又将是另一个面貌，但吞吐量更大，性能更好，稳定性更强，将是订单中心系统永远的追求。

- E N D -

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. [从零搭建一个 Spring Boot 开发环境](#)
2. [Redis 实现「附近的人」这个功能](#)
3. [一个秒杀系统的设计思考](#)
4. [零基础认识 Spring Boot](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章，点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！



作者 | 呼延十

链接 | juejin.im/post/5d351303f265da1bd30596f9

前言

本文主要受众为开发人员,所以不涉及到MySQL的服务部署等操作,且内容较多,大家准备好耐心和瓜子矿泉水.

前一阵系统的学习了一下MySQL,也有一些实际操作经验,偶然看到一篇和MySQL相关的面试文章,发现其中的一些问题自己也回答不好,虽然知识点大部分都知道,但是无法将知识串联起来.

因此决定搞一个MySQL灵魂100问,试着用回答问题的方式,让自己对知识点的理解更加深入一点.

此文不会事无巨细的从select的用法开始讲解mysql,主要针对的是开发人员需要知道的一些MySQL的知识点,主要包括索引,事务,优化等方面,以在面试中高频的问句形式给出答案.

1. 什么是索引?

索引是一种数据结构,可以帮助我们快速的进行数据的查找.

2. 索引是个什么样的数据结构呢?

索引的数据结构和具体存储引擎的实现有关, 在MySQL中使用较多的索引有Hash索引,B+树索引等,而我们经常使用的InnoDB存储引擎的默认索引实现为:B+树索引.

3. Hash索引和B+树所有有什么区别或者说优劣呢?

首先要知道Hash索引和B+树索引的底层实现原理:

hash索引底层就是hash表,进行查找时,调用一次hash函数就可以获取到相应的键值,之后进行回表查询获得实际数据.B+树底层实现是多路平衡查找树.对于每一次的查询都是从根节点出发,查找到叶子节点方可以获得所查键值,然后根据查询判断是否需要回表查询数据.

那么可以看出他们有以下的不同:

- hash索引进行等值查询更快(一般情况下),但是却无法进行范围查询.

因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.而B+树的所有节点皆遵

循(左节点小于父节点,右节点大于父节点,多叉树也类似),天然支持范围.

- hash索引不支持使用索引进行排序,原理同上.
- hash索引不支持模糊查询以及多列索引的最左前缀匹配.原理也是因为hash函数的不可预测AAAA和AAAB的索引没有相关性.
- hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.
- hash索引虽然在等值查询上较快,但是不稳定.性能不可预测,当某个键值存在大量重复的时候,发生hash碰撞,此时效率可能极差.而B+树的查询效率比较稳定,对于所有的查询都是从根节点到叶子节点,且树的高度较低.

因此,在大多数情况下,直接选择B+树索引可以获得稳定且较好的查询速度.而不需要使用hash索引.

4. 上面提到了B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据,什么是聚簇索引?

在B+树的索引中,叶子节点可能存储了当前的key值,也可能存储了当前的key值以及整行的数据,这就是聚簇索引和非聚簇索引. 在InnoDB中,只有主键索引是聚簇索引,如果没有主键,则挑选一个唯一键建立聚簇索引.如果没有唯一键,则隐式的生成一个键来建立聚簇索引.

当查询使用聚簇索引时,在对应的叶子节点,可以获取到整行数据,因此不用再次进行回表查询.

5. 非聚簇索引一定会回表查询吗?

不一定,这涉及到查询语句所要求的字段是否全部命中了索引,如果全部命中了索引,那么就不必再进行回表查询.

举个简单的例子,假设我们在员工表的年龄上建立了索引,那么当进行select age from employee where age < 20的查询时,在索引的叶子节点上,已经包含了age信息,不会再次进行回表查询.

6. 在建立索引的时候,都有哪些需要考虑的因素呢?

建立索引的时候一般要考虑到字段的使用频率,经常作为条件进行查询的字段比较适合.如果需要建立联合索引的话,还需要考虑联合索引中的顺序.此外也要考虑其他方面,比如防止过多的所有对表造成太大的压力.这些都和实际的表结构以及查询方式有关.

7. 联合索引是什么?为什么需要注意联合索引中的顺序?

MySQL可以使用多个字段同时建立一个索引,叫做联合索引.在联合索引中,如果想要命中索引,需要按照建立索引时的字段顺序挨个使用,否则无法命中索引.

具体原因为:

MySQL使用索引时需要索引有序,假设现在建立了"name,age,school"的联合索引,那么索引的排序为: 先按照name排序,如果name相同,则按照age排序,如果age的值也相等,则按照school进行排序.

当进行查询时,此时索引仅仅按照name严格有序,因此必须首先使用name字段进行等值查询,之后对于匹配到的列而言,其按照age字段严格有序,此时可以使用age字段用做索引查找,,,以此类推.因此在建立联合索引的时候应该注意索引列的顺序,一般情况下,将查询需求频繁或者字段选择性高的列放在前面.此外可以根据特例的查询或者表结构进行单独的调整.

8. 创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?

MySQL提供了explain命令来查看语句的执行计划,MySQL在执行某个语句之前,会将该语句过一遍查询优化器,之后会拿到对语句的分析,也就是执行计划,其中包含了许多信息. 可以通过其中和索引有关的信息来分析是否命中了索引,例如possible_key,key,key_len等字段,分别说明了此语句可能会使用的索引,实际使用的索引以及使用的索引长度.

9. 那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?

- 使用不等于查询,
- 列参与了数学运算或者函数
- 在字符串like时左边是通配符.类似于'%aaa'.
- 当mysql分析全表扫描比使用索引快的时候不使用索引.
- 当使用联合索引,前面一个条件为范围查询,后面的即使符合最左前缀原则,也无法使用索引.

以上情况,MySQL无法使用索引.

事务相关

1. 什么是事务?

理解什么是事务最经典的就是转账的栗子,相信大家也都了解,这里就不再说一边了.

事务是一系列的操作,他们要符合ACID特性.最常见的理解就是:事务中的操作要么全部成功,要么全部失败.但是只是这样还不够的.

2. ACID是什么?可以详细说一下吗?

A=Atomicity

原子性,就是上面说的,要么全部成功,要么全部失败.不可能只执行一部分操作.

C=Consistency

系统(数据库)总是从一个一致性的状态转移到另一个一致性的状态,不会存在中间状态.

I=Isolation

隔离性: 通常来说:一个事务在完全提交之前,对其他事务是不可见的.注意前面的通常来说加了红色,意味着有例外情况.

D=Durability

持久性,一旦事务提交,那么就永远是这样子了,哪怕系统崩溃也不会影响到这个事务的结果.

3. 同时有多个事务在进行会怎么样呢?

多事务的并发进行一般会造成以下几个问题:

- 脏读: A事务读取到了B事务未提交的内容,而B事务后面进行了回滚.
- 不可重复读: 当设置A事务只能读取B事务已经提交的部分,会造成在A事务内的两次查询,结果竟然不一样,因为在此期间B事务进行了提交操作.
- 幻读: A事务读取了一个范围的内容,而同时B事务在此期间插入了一条数据.造成"幻觉".

4. 怎么解决这些问题呢?MySQL的事务隔离级别了解吗?

MySQL的四种隔离级别如下:

- 未提交读(READ UNCOMMITTED)

这就是上面所说的例外情况了,这个隔离级别下,其他事务可以看到本事务没有提交的部分修改.因此会造成脏读的问题(读取到了其他事务未提交的部分,而之后该事务进行了回滚).

这个级别的性能没有足够大的优势,但是又有很多的问题,因此很少使用.

- 已提交读(READ COMMITTED)

其他事务只能读取到本事务已经提交的部分.这个隔离级别有 不可重复读的问题,在同一个事务内的两次读取,拿到的结果竟然不一样,因为另外一个事务对数据进行了修改.

- REPEATABLE READ(可重复读)

可重复读隔离级别解决了上面不可重复读的问题(看名字也知道),但是仍然有一个新问题,就是 幻读,当你读取id> 10 的数据行时,对涉及到的所有行加上了读锁,此时例外一个事务新插入了一条id=11的数据,因为是新插入的,所以不会触发上面的锁的排斥,那么进行本事务进行下一次的查询时会发现有一条id=11的数据,而上次的查询操作并没有获取到,再进行插入就会有主键冲突的问题.

- SERIALIZABLE(可串行化)

这是最高的隔离级别,可以解决上面提到的所有问题,因为他强制将所有的操作串行执行,这会导致并发性能极速下降,因此也不是很常用.

5. Innodb使用的是哪种隔离级别呢?

InnoDB默认使用的是可重复读隔离级别.

6. 对MySQL的锁了解吗?

当数据库有并发事务的时候,可能会产生数据的不一致,这时候需要一些机制来保证访问的次序,锁机制就是这样的一个机制.

就像酒店的房间,如果大家随意进出,就会出现多人抢夺同一个房间的情况,而在房间上装上锁,申请到钥匙的人才可以入住并且将房间锁起来,其他人只有等他使用完毕才可以再次使用.

7. MySQL都有哪些锁呢?像上面那样子进行锁定岂不是有点阻碍并发效率了?

从锁的类别上来讲,有共享锁和排他锁.

共享锁: 又叫做读锁. 当用户要进行数据的读取时,对数据加上共享锁.共享锁可以同时加上多个.

排他锁: 又叫做写锁. 当用户要进行数据的写入时,对数据加上排他锁.排他锁只可以加一个,他和其他的排他锁,共享锁都相斥.

用上面的例子来说就是用户的行为有两种,一种是来看房,多个用户一起看房是可以接受的. 一种是真正的入住一晚,在这期间,无论是想入住的还是想看房的都不可以.

锁的粒度取决于具体的存储引擎,InnoDB实现了行级锁,页级锁,表级锁.

他们的加锁开销从大大小,并发能力也是从大到小.

表结构设计

1. 为什么要尽量设定一个主键?

主键是数据库确保数据行在整张表唯一性的保障,即使业务上本张表没有主键,也建议添加一个自增长的ID列作为主键.设定了主键之后,在后续的删改查的时候可能更加快速以及确保操作数据范围安全.

2. 主键使用自增ID还是UUID?

推荐使用自增ID,不要使用UUID.

因为在InnoDB存储引擎中,主键索引是作为聚簇索引存在的,也就是说,主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序),如果主键索引是自增ID,那么只需要不断向后排列即可,如果是UUID,由于到来的ID与原来的大小不确定,会造成非常多的数据插入,数据移动,然后导致产生很多的内存碎片,进而造成插入性能的下降.

总之,在数据量大一些的情况下,用自增主键性能会好一些.

图片来源《高性能MySQL》： 其中默认后缀为使用自增ID，_uuid为使用UUID为主键的测试，测试了插入100w行和300w行的性能。

表 5-1：向InnoDB表插入数据的测试结果

表名	行数	时间（秒）	索引大小（MB）
userinfo	1 000 000	137	342
userinfo_uuid	1 000 000	180	544
userinfo	3 000 000	1233	1036
userinfo_uuid	3 000 000	4525	1707

关于主键是聚簇索引,如果没有主键,InnoDB会选择一个唯一键来作为聚簇索引,如果没有唯一键,会生成一个隐式的主键.

If you define a PRIMARY KEY on your table, InnoDB uses it as the clustered index.

If you do not define a PRIMARY KEY for your table, MySQL picks the first UNIQUE index that has only NOT NULL columns as the primary key and InnoDB uses it as the clustered index.

3. 字段为什么要求定义为not null?

MySQL官网这样介绍:

NULL columns require additional space in the row to record whether their values are NULL. For MyISAM tables, each NULL column takes one bit extra, rounded up to the nearest byte.

null值会占用更多的字节,且会在程序中造成很多与预期不符的情况.

4. 如果要存储用户的密码散列,应该使用什么字段进行存储?

密码散列,盐,用户身份证号等固定长度的字符串应该使用char而不是varchar来存储,这样可以节省空间且提高检索效率.

存储引擎相关

1. MySQL支持哪些存储引擎?

MySQL支持多种存储引擎,比如InnoDB,MyISAM,Memory,Archive等等.在大多数的情况下,直接选择使用InnoDB引擎都是最合适,InnoDB也是MySQL的默认存储引擎.

16. InnoDB和MyISAM有什么区别?

- InnoDB支持事物, 而MyISAM不支持事物
- InnoDB支持行级锁, 而MyISAM支持表级锁
- InnoDB支持MVCC, 而MyISAM不支持
- InnoDB支持外键, 而MyISAM不支持
- InnoDB不支持全文索引, 而MyISAM支持。

零散问题

1. MySQL中的varchar和char有什么区别.

char是一个定长字段,假如申请了char(10)的空间,那么无论实际存储多少内容.该字段都占用10个字符,而varchar是变长的,也就是说申请的只是最大长度,占用的空间为实际字符长度+1,最后一个字符存储使用了多长的空间.

在检索效率上来讲,char > varchar,因此在使用中,如果确定某个字段的值的长度,可以使用char,否则应该尽量使用varchar.例如存储用户MD5加密后的密码,则应该使用char.

2. varchar(10)和int(10)代表什么含义?

varchar的10代表了申请的空间长度,也是可以存储的数据的最大长度,而int的10只是代表了展示的长度,不足10位以0填充.也就是说,int(1)和int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示.

3. MySQL的binlog有几种录入格式?分别有什么区别?

有三种格式,statement,row和mixed.

- statement模式下,记录单元为语句.即每一个sql造成的影响会记录.由于sql的执行是有上下文的,因此在保存的时候需要保存相关的信息,同时还有一些使用了函数之类的语句无法被记录复制.
- row级别下,记录单元为每一行的改动,基本是可以全部记下来但是由于很多操作,会导致大量行的改动(比如alter table),因此这种模式的文件保存的信息太多,日志量太大.
- mixed. 一种折中的方案,普通操作使用statement记录,当无法使用statement的时候使用row.

此外,新版的MySQL中对row级别也做了一些优化,当表结构发生变化的时候,会记录语句而不是逐行记录.

4. 超大分页怎么处理?

超大的分页一般从两个方向上来解决.

- 数据库层面,这也是我们主要集中关注的(虽然收效没那么大),类似于select * from table where age > 20 limit 1000000,10这种查询其实也是有可以优化的余地的. 这条语句需要load1000000数据然后基本上全部丢弃,只取10条当然比较慢. 当时我们可以修改为select * from table where id in (select id from table where age > 20 limit 1000000,10)这样虽然也load了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快. 同时如果ID连续的好,我们还可以select * from table where id > 1000000 limit 10,效率也是不错的,优化的可能性有许多种,但是核心思想都一样,就是减少load的数据.
- 从需求的角度减少这种请求....主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止ID泄漏且连续被人恶意攻击.

解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至redis等k-v数据库中,直接返回即可.

在阿里巴巴《Java开发手册》中,对超大分页的解决办法是类似于上面提到的第一种.

7. 【推荐】利用延迟关联或者子查询优化超多分页场景。

说明 : MySQL 并不是跳过 offset 行 , 而是取 offset+N 行 , 然后返回放弃前 offset 行 , 返回 N 行 , 那当 offset 特别大的时候 , 效率就非常的低下 , 要么控制返回的总页数 , 要么对超过特定阈值的页数进行 SQL 改写。

正例 : 先快速定位需要获取的 id 段 , 然后再关联 :

```
SELECT a.* FROM 表1 a, (select id from 表1 where 条件 LIMIT 100000,20 ) b where a.id=b.id
```

5. 关心过业务系统里面的sql耗时吗?统计过慢查询吗?对慢查询都怎么优化过?

在业务系统中,除了使用主键进行的查询,其他的我都会在测试库上测试其耗时,慢查询的统计主要由运维在做,会定期将业务中的慢查询反馈给我们.

慢查询的优化首先要搞明白慢的原因是什么?是查询条件没有命中索引?是load了不需要的数据列?还是数据量太大?

所以优化也是针对这三个方向来的,

- 首先分析语句,看看是否load了额外的数据,可能是查询了多余的行并且抛弃掉了,可能是加载了许多结果中并不需要的列,对语句进行分析以及重写.
- 分析语句的执行计划,然后获得其使用索引的情况,之后修改语句或者修改索引,使得语句可以尽可能的命中索引.
- 如果对语句的优化已经无法进行,可以考虑表中的数据量是否太大,如果是的话可以进行横向或者纵向的分表.

6. 上面提到横向分表和纵向分表,可以分别举一个适合他们的例子吗?

横向分表是按行分表.假设我们有一张用户表,主键是自增ID且同时是用户的ID.数据量较大,有1亿多条,那么此时放在一张表里的查询效果就不太理想.我们可以根据主键ID进行分表,无论是按尾号分,或者按ID的区间分都是可以的.假设按照尾号0-99分为100个表,那么每张表中的数据就仅有100w.这时的查询效率无疑是可以满足要求的.

纵向分表是按列分表.假设我们现在有一张文章表.包含字段id-摘要-内容.而系统中的展示形式是刷新出一个列表,列表中仅包含标题和摘要,当用户点击某篇文章进入详情时才需要正文内容.此时,如果数据量大,将内容这个很大且不经常使用的列放在一起会拖慢原表的查询速度.我们可以将上面的表分为两张.id-摘要,id-内容.当用户点击详情,那主键再来取一次内容即可.而增加的存储量只是很小的主键字段.代价很小.

当然,分表其实和业务的关联度很高,在分表之前一定要做好调研以及benchmark.不要按照自己的猜想盲目操作.

7. 什么是存储过程?有哪些优缺点?

存储过程是一些预编译的SQL语句。1、更加直白的理解：存储过程可以说是一个记录集，它是由一些T-SQL语句组成的代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。2、存储过程是一个预编译的代码块，执行效率比较高，一个存储过程替代大量T_SQL语句，可以降低网络通信量，提高通信速率，可以一定程度上确保数据安全

但是,在互联网项目中,其实是不太推荐存储过程的,比较出名的就是阿里的《Java开发手册》中禁止使用存储过程,我个人的理解是在互联网项目中,迭代太快,项目的生命周期也比较短,人员流动相比于传统的项目也更加频繁,在这样的情况下,存储过程的管理确实是没有那么方便,同时,复用性也没有写在服务层那么好.

8. 说一说三个范式

第一范式: 每个列都不可以再拆分. 第二范式: 非主键列完全依赖于主键,而不能是依赖于主键的一部分. 第三范式: 非主键列只依赖于主键,不依赖于其他非主键.

在设计数据库结构的时候,要尽量遵守三范式,如果不遵守,必须有足够的理由.比如性能.事实上我们经常会为了性能而妥协数据库的设计.

9. MyBatis中的#和\$有什么区别?

乱入了一个奇怪的问题.....我只是想单独记录一下这个问题,因为出现频率太高了.

会将传入的内容当做字符串,而\$会直接将传入值拼接在sql语句中.

所以#可以在一定程度上预防sql注入攻击.

tips: 大家可以关注微信公众号: Java后端, 获取更多优秀博文推送。

-END-

如果看到这里,说明你喜欢这篇文章,请[转发、点赞](#)。微信搜索「web_resource」,关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. 感受 Lambda 之美,推荐收藏,需要时查阅
2. 如何优雅的导出 Excel
3. 文艺互联网公司 vs 二逼互联网公司
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看

MySQL: Left Join 避坑指南

Java后端 2019-12-08

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | MageekChiu

链接 | segmentfault.com/a/1190000020458807

现象

left join在我们使用mysql查询的过程中可谓非常常见，比如博客里一篇文章有多少条评论、商城里一个货物有多少评论、一条评论有多少个赞等等。但是由于对join、on、where等关键字的不熟悉，有时候会导致查询结果与预期不符，所以今天我就来总结一下，一起避坑。

这里我先给出一个场景，并抛出两个问题，如果你都能答对那这篇文章就不用看了。

假设有一个班级管理应用，有一个表classes，存了所有的班级；有一个表students，存了所有的学生，具体数据如下（感谢廖雪峰的在线SQL）：

SELECT * FROM classes;

id	name
1	一班
2	二班
3	三班
4	四班

SELECT * FROM students;

id	class_id	name	gender
1	1	小明	M
2	1	小红	F
3	1	小军	M
4	1	小米	F
5	2	小白	F
6	2	小兵	M
7	2	小林	M
8	3	小新	F
9	3	小王	M
10	3	小丽	F

那么现在有两个需求：

- 找出每个班级的名称及其对应的女同学数量
- 找出一班的同学总数

对于需求1，大多数人不假思索就能想出如下两种sql写法，请问哪种是对的？

```
SELECT c.name, count(s.name) as num
FROM classes c left join students s
on s.class_id = c.id
and s.gender = 'F'
group by c.name
```

或者

```
SELECT c.name, count(s.name) as num
FROM classes c left join students s
on s.class_id = c.id
where s.gender = 'F'
group by c.name
```

对于需求2，大多数人也可以不假思索的想出如下两种sql写法，请问哪种是对的？

```
SELECT c.name, count(s.name) as num
FROM classes c left join students s
on s.class_id = c.id
where c.name = '一班'
group by c.name
```

或者

```
SELECT c.name, count(s.name) as num
FROM classes c left join students s
on s.class_id = c.id
and c.name = '一班'
group by c.name
```

请不要继续往下翻！！先给出你自己的答案，正确答案就在下面。

答案是两个需求都是第一条语句是正确的，要搞清楚这个问题，就得明白mysql对于left join的执行原理，下节进行展开。

根源

mysql 对于left join的采用类似嵌套循环的方式来进行从处理，以下面的语句为例：

```
SELECT * FROM LT LEFT JOIN RT ON P1(LT,RT)) WHERE P2(LT,RT)
```

其中P1是on过滤条件，缺失则认为是TRUE，P2是where过滤条件，缺失也认为是TRUE，该语句的执行逻辑可以描述为：

```
FOR each row lt in LT { // 遍历左表的每一行
    BOOL b = FALSE;
    FOR each row rt in RT such that P1(lt, rt) { // 遍历右表每一行，找到满足join条件的行
        IF P2(lt, rt) { // 满足 where 过滤条件
            t:=lt||rt; // 合并行，输出该行
        }
        b=TRUE; // lt在RT中有对应的行
    }
    IF (!b) { // 遍历完RT，发现lt在RT中没有有对应的行，则尝试用null补一行
        IF P2(lt,NULL) { // 补上null后满足 where 过滤条件
            t:=lt||NULL; // 输出lt和null补上的行
        }
    }
}
```

当然，实际情况中MySQL会使用buffer的方式进行优化，减少行比较次数，不过这不影响关键的执行流程，不在本文讨论范围内。

从这个伪代码中，我们可以看出两点：

如果想对右表进行限制，则一定要在on条件中进行，若在where中进行则可能导致数据缺失，导致左表在右表中无匹配行的行在最终结果中不出现，违背了我们对left join的理解。因为对左表无右表匹配行的行而言，遍历右表后b=FALSE,所以会尝试用NULL补齐右表，但是此时我们的P2对右表行进行了限制，NULL若不满足P2(NULL一般都不会满足限制条件，除非IS NULL这种)，则不会加入最终的结果中，导致结果缺失。

如果没有where条件，无论on条件对左表进行怎样的限制，左表的每一行都至少会有一行的合成结果，对左表行而言，若右表若没有对应的行，则右表遍历结束后b=FALSE，会用一行NULL来生成数据，而这个数据是多余的。所以对左表进行过滤必须用where。

下面展开两个需求的错误语句的执行结果和错误原因：

需求1

name	num
一班	2
二班	1
三班	2

需求2

name	num
一班	4
二班	0
三班	0
四班	0

需求1由于在where条件中对右表限制，导致数据缺失（四班应该有个为0的结果）

需求2由于在on条件中对左表限制，导致数据多余（其他班的结果也出来了，还是错的）

总结

通过上面的问题现象和分析，可以得出了结论：在left join语句中，左表过滤必须放where条件中，右表过滤必须放on条件中，这样结果才能不多不少，刚刚好。

SQL 看似简单，其实也有很多细节原理在里面，一个小小的混淆就会造成结果与预期不符，所以平时要注意这些细节原理，避免关键时候出错。

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. Spring Boot:启动原理解析
2. 一键下载 Pornhub 视频！
3. Spring Boot 多模块项目实践(附打包方法)
4. 一个女生不主动联系你还有机会吗？
5. 团队开发中 Git 最佳实践



[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

MySQL：数据库优化，可以看看这篇文章

Java后端 2019-11-21

作者 | 赵栩彬

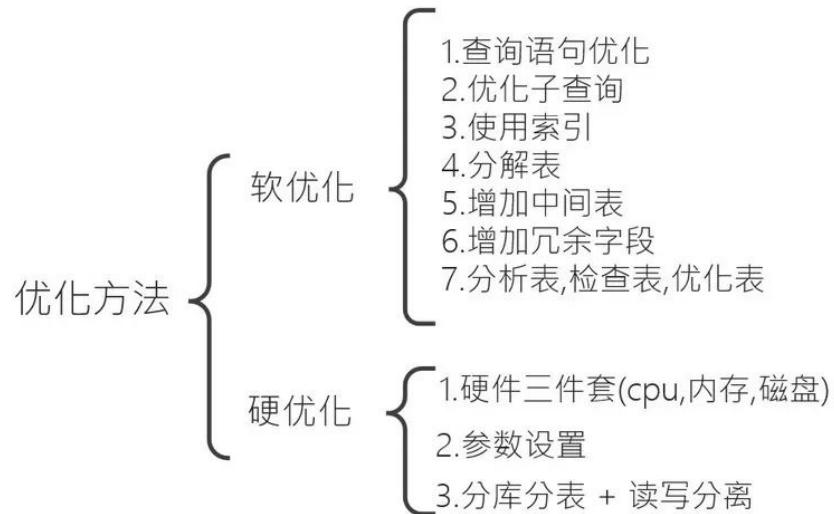
链接 | segmentfault.com/a/1190000018631870

上篇 | [一千行 MySQL 学习笔记](#)

前言

数据库优化一方面是找出系统的瓶颈,提高MySQL数据库的整体性能,而另一方面需要合理的结构设计和参数调整,以提高用户的响应速度,同时还要尽可能的节约系统资源,以便让系统提供更大的负荷.

1. 优化一览图



2. 优化

笔者将优化分为了两大类,软优化和硬优化,软优化一般是操作数据库即可,而硬优化则是操作服务器硬件及参数设置.

2.1 软优化

2.1.1 查询语句优化

1.首先我们可以用EXPLAIN或DESCRIBE(简写:DESC)命令分析一条查询语句的执行信息.

2.例:

```
DESC SELECT * FROM `user`
```

显示:

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	user	ALL	(Null)	(Null)	(Null)	(Null)	1	(Null)

其中会显示索引和查询数据读取数据条数等信息.

2.1.2 优化子查询

在MySQL中,尽量使用JOIN来代替子查询.因为子查询需要嵌套查询,嵌套查询时会建立一张临时表,临时表的建立和删除都会有较大的系统开销,而连接查询不会创建临时表,因此效率比嵌套子查询高.

2.1.3 使用索引

索引是提高数据库查询速度最重要的方法之一,关于索引可以参高笔者一文,介绍比较详细,此处记录使用索引的三大注意事项:

- LIKE关键字匹配 '%'开头的字符串,不会使用索引.
- OR关键字的两个字段必须都是用了索引,该查询才会使用索引.
- 使用多列索引必须满足最左匹配.

2.1.4 分解表

对于字段较多的表,如果某些字段使用频率较低,此时应当,将其分离出来从而形成新的表,

2.1.5 中间表

对于将大量连接查询的表可以创建中间表,从而减少在查询时造成的连接耗时.

2.1.6 增加冗余字段

类似于创建中间表,增加冗余也是为了减少连接查询.

2.1.7 分析表,检查表,优化表

分析表主要是分析表中关键字的分布,检查表主要是检查表中是否存在错误,优化表主要是消除删除或更新造成的表空间浪费.

1. 分析表: 使用 ANALYZE 关键字,如ANALYZE TABLE user;

Table	Op	Msg_type	Msg_text
► iot.user	analyze	status	OK

- Op:表示执行的操作.
- Msg_type:信息类型,有status,info,note,warning,error.
- Msg_text:显示信息.

2. 检查表: 使用 CHECK关键字,如CHECK TABLE user [option]

option 只对MyISAM有效,共五个参数值:

- QUICK:不扫描行,不检查错误的连接.
- FAST:只检查没有正确关闭的表.
- CHANGED:只检查上次检查后被更改的表和没被正确关闭的表.
- MEDIUM:扫描行,以验证被删除的连接是有效的,也可以计算各行关键字校验和.
- EXTENDED:最全面的的检查,对每行关键字全面查找.

3. 优化表:使用OPTIMIZE关键字,如OPTIMIZE [LOCAL|NO_WRITE_TO_BINLOG] TABLE user;

LOCAL|NO_WRITE_TO_BINLOG都是表示不写入日志.,优化表只对VARCHAR,BLOB和TEXT有效,通过OPTIMIZE TABLE语句可以

2.2 硬优化

2.2.1 硬件三件套

1. 配置多核心和频率高的cpu,多核心可以执行多个线程.
2. 配置大内存,提高内存,即可提高缓存区容量,因此能减少磁盘I/O时间,从而提高响应速度.
3. 配置高速磁盘或合理分布磁盘:高速磁盘提高I/O,分布磁盘能提高并行操作的能力.

2.2.2 优化数据库参数

优化数据库参数可以提高资源利用率,从而提高MySQL服务器性能.MySQL服务的配置参数都在my.cnf或my.ini,下面列出性能影响较大的几个参数.

- key_buffer_size:索引缓冲区大小
- table_cache:能同时打开表的个数
- query_cache_size和query_cache_type:前者是查询缓冲区大小,后者是前面参数的开关,0表示不使用缓冲区,1表示使用缓冲区,但可以在查询中使用SQL_NO_CACHE表示不要使用缓冲区,2表示在查询中明确指出使用缓冲区才用缓冲区,即SQL_CACHE.
- sort_buffer_size:排序缓冲区

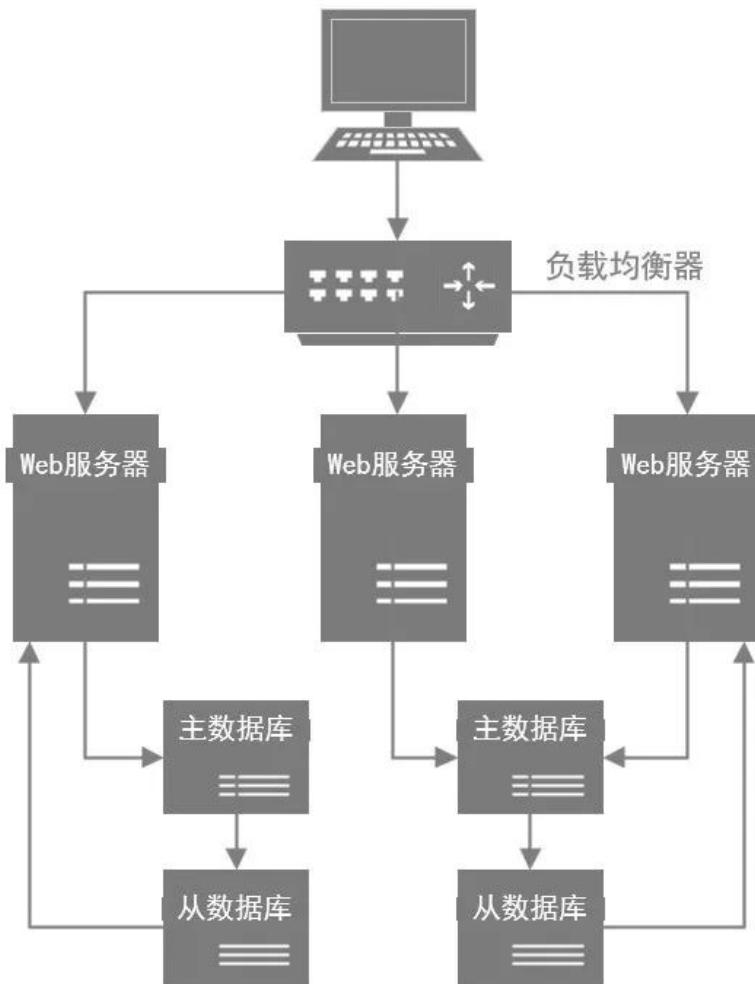
更多参数传送门:

<https://www.mysql.com/cn/why-mysql/performance/index.html>

2.2.3 分库分表

因为数据库压力过大,首先一个问题就是高峰期系统性能可能会降低,因为数据库负载过高对性能会有影响.另外一个,压力过大把你的数据库给搞挂了怎么办?

所以此时你必须得对系统做分库分表 + 读写分离,也就是把一个库拆分为多个库,部署在多个数据库服务上,这时作为主库承载写入请求.然后每个主库都挂载至少一个从库,由从库来承载读请求.



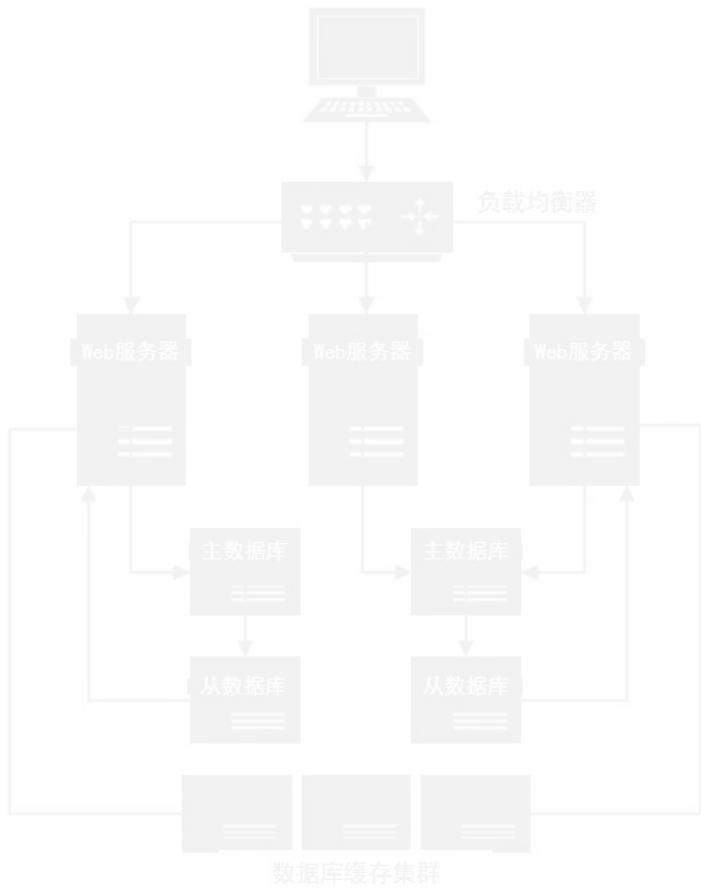
2.2.4 缓存集群

如果用户量越来越大，此时你可以不停的加机器，比如说系统层面不停加机器，就可以承载更高的并发请求。然后数据库层面如果写入并发越来越高，就扩容加数据库服务器，通过分库分表是可以支持扩容机器的，如果数据库层面的读并发越来越高，就扩容加更多的从库。

但是这里有一个很大的问题：数据库其实本身不是用来承载高并发请求的，所以通常来说，数据库单机每秒承载的并发就在几千的数量级，而且数据库使用的机器都是比较高配置，比较昂贵的机器，成本很高。如果你就是简单的不停的加机器，其实是不对的。**所以在高并发架构里通常都有缓存这个环节，缓存系统的设计就是为了承载高并发而生。**

所以单机承载的并发量都在每秒几万，甚至每秒数十万，对高并发的承载能力比数据库系统要高出一到两个数量级。所以你完全可以根据系统的业务特性，对那种写少读多的请求，引入缓存集群。

具体来说，就是在写数据库的时候同时写一份数据到缓存集群里，然后用缓存集群来承载大部分的读请求。这样的话，通过缓存集群，就可以用更少的机器资源承载更高的并发。



结语

一个完整而复杂的高并发系统架构中，一定会包含：各种复杂的自研基础架构系统。各种精妙的架构设计.因此一篇小文顶多具有抛砖引玉的效果,但是数据库优化的思想差不多就这些了.

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. [一千行 MySQL 学习笔记](#)
2. [一份工作坚持多久跳槽最合适?](#)
3. [项目中常用到的 19 条 MySQL 优化](#)
4. [零基础认识 Spring Boot](#)
5. [团队开发中 Git 最佳实践](#)



声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

【小技巧】MyBatis 中 SQL 写法技巧小总结

koala Java后端 2月27日



最近有个兄弟在搞mybatis，问我怎么写sql，说简单一点mybatis就是写原生sql，官方都说了 mybatis 的动态sql语句是基于OGNL表达式的。可以方便的在 sql 语句中实现某些逻辑。总体说来mybatis 动态SQL 语句主要有以下几类：

1. if 语句 (简单的条件判断)
2. choose (when,otherwise) ,相当于java 语言中的 switch ,与 jstl 中的choose 很类似.
3. trim (对包含的内容加上 prefix,或者 suffix 等, 前缀, 后缀)
4. where (主要是用来简化sql语句中where条件判断的, 能智能的处理 and or ,不必担心多余导致语法错误)
5. set (主要用于更新时)
6. foreach (在实现 mybatis in 语句查询时特别有用)

我说一下：if when 都仅仅对于Map类型才能进行判断，Integer, String 那些都不能进行判断，虽然说mybatis最后都是把参数封装为一个Map集合再通过占位符接入的。另一个就是trim很强大，可以说where和set能干的他都能干。choose在进行”单个“模糊查询时候很方便。

往期优质技术文章可以关注微信公众号：Java后端，后台回复 技术博文 获取。

下面分别介绍这几种处理方式

1. mybatis if 语句处理

```
<select id="dynamicIfTest" parameterType="Blog" resultType="Blog">
    select * from t_blog where 1 = 1
    <if test="title != null">
        and title = #{title}
    </if>
    <if test="content != null">
        and content = #{content}
    </if>
    <if test="owner != null">
        and owner = #{owner}
    </if>
</select>
```

这条语句的意思非常简单，如果你提供了title参数，那么就要满足title=#{title}，同样如果你提供了Content和Owner的时候，它们也需要满足相应的条件，之后就是返回满足这些条件的所有Blog，这是非常有用的一个功能，以往我们使用其他类型框架或者直接使用JDBC的时候，如果我们要达到同样的选择效果的时候，我们就需要拼SQL语句，这是极其麻烦的，比起来，上述的动态SQL就要简单多了。

2. choose (when,otherwise) ,相当于java 语言中的 switch ,与 jstl 中的choose 很类似

```

<select id="dynamicChooseTest" parameterType="Blog" resultType="Blog">
    select * from t_blog where 1 = 1
    <choose>
        <when test="title != null">
            and title = #{title}
        </when>
        <when test="content != null">
            and content = #{content}
        </when>
        <otherwise>
            and owner = "owner1"
        </otherwise>
    </choose>
</select>

```

when元素表示当when中的条件满足的时候就输出其中的内容，跟JAVA中的switch效果差不多的是按照条件的顺序，当when中有条件满足的时候，就会跳出choose，即所有的when和otherwise条件中，只有一个会输出，当所有的我很条件都不满足的时候就输出otherwise中的内容。所以上述语句的意思非常简单，当title!=null的时候就输出and title = #{title}，不再往下判断条件，当title为空且content!=null的时候就输出and content = #{content}，当所有条件都不满足的时候就输出otherwise中的内容。

3.trim (对包含的内容加上 prefix,或者 suffix 等，前缀，后缀)

```

<select id="dynamicTrimTest" parameterType="Blog" resultType="Blog">
    select * from t_blog
    <trim prefix="where" prefixOverrides="and |or">
        <if test="title != null">
            title = #{title}
        </if>
        <if test="content != null">
            and content = #{content}
        </if>
        <if test="owner != null">
            or owner = #{owner}
        </if>
    </trim>
</select>

```

后缀，与之对应的属性是prefix和suffix；可以把包含内容的首部某些内容覆盖，即忽略，也可以把尾部的某些内容覆盖，对应的属性是prefixOverrides和suffixOverrides；正因为trim有这样的功能，所以我们也可以非常的简单的利用trim来代替where元素的功能。

4. where (主要是用来简化sql语句中where条件判断的，能智能的处理 and or 条件)

```

<select id="dynamicWhereTest" parameterType="Blog" resultType="Blog">
    select * from t_blog
    <where>
        <if test="title != null">
            title = #{title}
        </if>
        <if test="content != null">
            and content = #{content}
        </if>
        <if test="owner != null">
            and owner = #{owner}
        </if>
    </where>
</select>

```

where元素的作用是会在写入where元素的地方输出一个where，另外一个好处是你不需要考虑where元素里面的条件输出是什么样子的，MyBatis会智能的帮你处理，如果所有的条件都不满足那么MyBatis就会查出所有的记录，如果输出后是and 开头的，MyBatis会把第一个and忽略，当然如果是or开头的，MyBatis也会把它忽略；此外，在where元素中你不需要考虑空格的问题，MyBatis会智能的帮你加上。像上述例子中，如果title=null，而content != null，那么输出的整个语句会是select * from t_blog where content = #{content}，而不是select * from t_blog where and content = #{content}，因为MyBatis会智能的把首个and 或 or 给忽略。

5.set (主要用于更新时)

```

<update id="dynamicSetTest" parameterType="Blog">
    update t_blog
    <set>
        <if test="title != null">
            title = #{title},
        </if>
        <if test="content != null">
            content = #{content},
        </if>
        <if test="owner != null">
            owner = #{owner}
        </if>
    </set>
    where id = #{id}
</update>

```

set元素主要是用在更新操作的时候，它的主要功能和where元素其实是差不多的，主要是在包含的语句前输出一个set，然后如果包含的语句是以逗号结束的话将会把该逗号忽略，如果set包含的内容为空的话则会出错。有了set元素我们就可以动态的更新那些修改了的字段。

6. foreach (在实现 mybatis in 语句查询时特别有用)

foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。foreach元素的属性主要有item, index, collection, open, separator, close。item表示集合中每一个元素进行迭代时的别名，index指定一个名字，用于表示在迭代过程中，每次迭代到的位置，open表示该语句以什么开始，separator表示在每次进行迭代之间以什么符号作为分隔符，close表示以什么结束，在使用foreach的时候最关键的也是最容易出错的就是collection属性，该属性是必须指定的，但是在不同情况下，该属性的值是不一样的，主要有一下3种情况：

- 如果传入的是单参数且参数类型是一个List的时候，collection属性值为list。
- 如果传入的是单参数且参数类型是一个array数组的时候，collection的属性值为array。
- 如果传入的参数是多个的时候，我们就需要把它们封装成一个Map了，当然单参数也可以封装成map，实际上如果你在传入参数的时候，在MyBatis里面也是会把它封装成一个Map的，map的key就是参数名，所以这个时候collection属性值就是传入的List或array对象在自己封装的map里面的key。

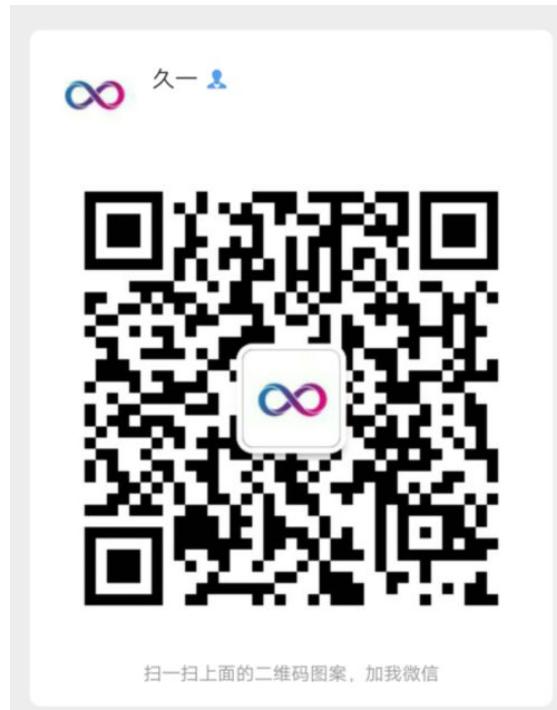
作者：牧羊人影视

原文链接：<https://blog.csdn.net/tengxing007/article/details/54864897>

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓ 扫描二维码添加小编 ↓



推荐阅读

1. 狠！删库跑路！
2. 分布式与集群的区别究竟是什么？
3. 看完这篇 HTTP，面试官就难不倒你了
4. 快速建站利器！



微信搜一搜

Q Java后端

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

一千行 MySQL 学习笔记

格物 Java后端 2019-11-20

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

MySQL命令和语句挺多, 全部记忆下来不现实, 况且有不常用的指令。下面把大部分的指令做了记录和详细的注释。建议收藏、转发此篇文章, 如果忘记可以翻出来查查。也同样欢迎围观作者的博客: shockerli.net

Tips: 手机浏览代码可以左右滑动, 建议转发此文章。

Windows服务

```
1 -- 启动MySQL
2 net start mysql
3 -- 创建Windows服务
4 sc create mysql binPath= mysqld_bin_path(注意: 等号与值之间有空格)
```

连接与断开服务器

```
1 mysql -h 地址 -P 端口 -u 用户名 -p 密码
2
3 SHOW PROCESSLIST -- 显示哪些线程正在运行
4 SHOW VARIABLES -- 显示系统变量信息
```

数据库操作

```
1 -- 查看当前数据库
2 SELECT DATABASE();
3 -- 显示当前时间、用户名、数据库版本
4 SELECT now(), user(), version();
5 -- 创建库
6 CREATE DATABASE[ IF NOT EXISTS] 数据库名 数据库选项
    数据库选项:
8     CHARACTER SET charset_name
9     COLLATE collation_name
10 -- 查看已有库
11 SHOW DATABASES[ LIKE 'PATTERN']
12 -- 查看当前库信息
13 SHOW CREATE DATABASE 数据库名
14 -- 修改库的选项信息
15 ALTER DATABASE 库名 选项信息
16 -- 删除库
17 DROP DATABASE[ IF EXISTS] 数据库名
    同时删除该数据库相关的目录及其目录内容
```

表的操作

```
1 -- 创建表
2 CREATE [TEMPORARY] TABLE[ IF NOT EXISTS] [库名.]表名 ( 表的结构定义 )[ 表选项]
```

每个字段必须有数据类型
最后一个字段后不能有逗号
TEMPORARY 临时表，会话结束时表自动消失
对于字段的定义：
字段名 数据类型 [NOT NULL | NULL] [DEFAULT default_value] [AUTO_INCREMENT] [UNIQUE [KEY]] | [PRIMARY] KEY

-- 表选项
-- 字符集
CHARSET = charset_name
如果表没有设定，则使用数据库字符集
-- 存储引擎
ENGINE = engine_name
表在管理数据时采用的不同的数据结构，结构不同会导致处理方式、提供的特性操作等不同
常见的引擎：InnoDB MyISAM Memory/Heap BDB Merge Example CSV MaxDB Archive
不同的引擎在保存表的结构和数据时采用不同的方式
MyISAM表文件含义：.frm表定义，.MYD表数据，.MYI表索引
InnoDB表文件含义：.frm表定义，表空间数据和日志文件
SHOW ENGINES -- 显示存储引擎的状态信息
SHOW ENGINE 引擎名 {LOGS|STATUS} -- 显示存储引擎的日志或状态信息
-- 自增起始数
AUTO_INCREMENT = 行数
-- 数据文件目录
DATA DIRECTORY = '目录'
-- 索引文件目录
INDEX DIRECTORY = '目录'
-- 表注释
COMMENT = 'string'
-- 分区选项
PARTITION BY ... (详细见手册)
-- 查看所有表
SHOW TABLES[LIKE 'pattern']
SHOW TABLES FROM 表名
-- 查看表机构
SHOW CREATE TABLE 表名 (信息更详细)
DESC 表名 / DESCRIBE 表名 / EXPLAIN 表名 / SHOW COLUMNS FROM 表名 [LIKE 'PATTERN']
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']
-- 修改表
-- 修改表本身的选项
ALTER TABLE 表名 表的选项
eg: ALTER TABLE 表名 ENGINE=MYISAM;
-- 对表进行重命名
RENAME TABLE 原表名 TO 新表名
RENAME TABLE 原表名 TO 库名.表名 (可将表移动到另一个数据库)
-- RENAME可以交换两个表名
-- 修改表的字段机构 (13.1.2. ALTER TABLE语法)
ALTER TABLE 表名 操作名
-- 操作名
ADD[COLUMN] 字段定义 -- 增加字段
AFTER 字段名 -- 表示增加在该字段名后面
FIRST -- 表示增加在第一个
ADD PRIMARY KEY(字段名) -- 创建主键
ADD UNIQUE [索引名] (字段名)-- 创建唯一索引
ADD INDEX [索引名] (字段名) -- 创建普通索引
DROP[COLUMN] 字段名 -- 删除字段
MODIFY[COLUMN] 字段名 字段属性 -- 支持对字段属性进行修改，不能修改字段名(所有原有属性也需写上)

```
57 CHANGE[ COLUMN] 原字段名 新字段名 字段属性      -- 支持对字段名修改
58 DROP PRIMARY KEY      -- 删除主键(删除主键前需删除其AUTO_INCREMENT属性)
59 DROP INDEX 索引名 -- 删除索引
60 DROP FOREIGN KEY 外键      -- 删除外键
61 -- 删表
62   DROP TABLE[ IF EXISTS] 表名 ...
63 -- 清空表数据
64   TRUNCATE [TABLE] 表名
65 -- 复制表结构
66   CREATE TABLE 表名 LIKE 要复制的表名
67 -- 复制表结构和数据
68   CREATE TABLE 表名 [AS] SELECT * FROM 要复制的表名
69 -- 检查表是否有错误
70   CHECK TABLE tbl_name [,tbl_name] ... [option] ...
71 -- 优化表
72   OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [,tbl_name] ...
73 -- 修复表
74   REPAIR [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [,tbl_name] ... [QUICK] [EXTENDED] [USE_FRM]
75 -- 分析表
76   ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [,tbl_name] ...
```



数据操作

```
1 -- 增
2   INSERT [INTO] 表名 [(字段列表)] VALUES (值列表)[, (值列表), ...]
3     -- 如果要插入的值列表包含所有字段并且顺序一致，则可以省略字段列表。
4     -- 可同时插入多条数据记录！
5     REPLACE 与 INSERT 完全一样，可互换。
6   INSERT [INTO] 表名 SET 字段名=值[, 字段名=值, ...]
7 -- 查
8   SELECT 字段列表 FROM 表名[ 其他子句]
9     -- 可来自多个表的多个字段
10    -- 其他子句可以不使用
11    -- 字段列表可以用*代替，表示所有字段
12 -- 删
13   DELETE FROM 表名[ 删除条件子句]
14     没有条件子句，则会删除全部
15 -- 改
16   UPDATE 表名 SET 字段名=新值[, 字段名=新值] [更新条件]
```

字符集编码

```
1 -- MySQL、数据库、表、字段均可设置编码
2 -- 数据编码与客户端编码不需一致
3 SHOW VARIABLES LIKE 'character_set_%'    -- 查看所有字符集编码项
4   character_set_client      客户端向服务器发送数据时使用的编码
5   character_set_results      服务器端将结果返回给客户端所使用的编码
6   character_set_connection    连接层编码
7 SET 变量名 = 变量值
8   SET character_set_client = gbk;
9   SET character_set_results = gbk;
10  SET character_set_connection = gbk;
11 SET NAMES GBK;    -- 相当于完成以上三个设置
```

```
12 -- 校对集
13   校对集用以排序
14   SHOW CHARACTER SET [LIKE 'pattern']/SHOW CHARSET [LIKE 'pattern']    查看所有字符集
15   SHOW COLLATION [LIKE 'pattern']      查看所有校对集
16   CHARSET 字符集编码      设置字符集编码
17   COLLATE 校对集编码      设置校对集编码
```

数据类型（列类型）

```
1  1. 数值类型
2
3  -- a. 整型 -----
4    类型      字节      范围（有符号位）
5    tinyint    1字节    -128 ~ 127      无符号位：0 ~ 255
6    smallint   2字节    -32768 ~ 32767
7    mediumint  3字节    -8388608 ~ 8388607
8    int        4字节
9    bigint     8字节
10   int(M)   M表示总位数
11   - 默认存在符号位，unsigned 属性修改
12   - 显示宽度，如果某个数不够定义字段时设置的位数，则前面以0补填，zerofill 属性修改
13     例：int(5) 插入一个数'123'，补填后为'00123'
14   - 在满足要求的情况下，越小越好。
15   - 1表示bool值真，0表示bool值假。MySQL没有布尔类型，通过整型0和1表示。常用tinyint(1)表示布尔型。
16
17  -- b. 浮点型 -----
18    类型      字节      范围
19    float(单精度) 4字节
20    double(双精度) 8字节
21    浮点型既支持符号位 unsigned 属性，也支持显示宽度 zerofill 属性。
22    不同于整型，前后均会补填0。
23    定义浮点型时，需指定总位数和小数位数。
24      float(M, D)    double(M, D)
25      M表示总位数，D表示小数位数。
26      M和D的大小会决定浮点数的范围。不同于整型的固定范围。
27      M既表示总位数（不包括小数点和正负号），也表示显示宽度（所有显示符号均包括）。
28      支持科学计数法表示。
29      浮点数表示近似值。
30
31  -- c. 定点数 -----
32    decimal -- 可变长度
33    decimal(M, D)  M也表示总位数，D表示小数位数。
34    保存一个精确的数值，不会发生数据的改变，不同于浮点数的四舍五入。
35    将浮点数转换为字符串来保存，每9位数字保存为4个字节。
36
37  2. 字符串类型
38
39  -- a. char, varchar -----
40    char    定长字符串，速度快，但浪费空间
41    varchar 变长字符串，速度慢，但节省空间
42    M表示能存储的最大长度，此长度是字符数，非字节数。
43    不同的编码，所占用的空间不同。
44    char,最多255个字符，与编码无关。
45    varchar 最多65535字符，与编码有关。
```

45 varchar, 最多65535字符, 与编码有关。
46 一条有效记录最大不能超过65535个字节。
47 utf8 最大为21844个字符, gbk 最大为32766个字符, latin1 最大为65532个字符
48 varchar 是变长的, 需要利用存储空间保存 varchar 的长度, 如果数据小于255个字节, 则采用一个字节来保存长度, 反之需要两个字节来保存。
49 varchar 的最大有效长度由最大行大小和使用的字符集确定。
50 最大有效长度是65532字节, 因为在varchar存字符串时, 第一个字节是空的, 不存在任何数据, 然后还需两个字节来存放字符串的长度, 所以有效
51 例: 若一个表定义为 CREATE TABLE tb(c1 int, c2 char(30), c3 varchar(N)) charset=utf8; 问N的最大值是多少? 答: (65535-2)/2=32766
52

-- b. blob, text -----
blob 二进制字符串 (字节字符串)
tinyblob, blob, mediumblob, longblob
text 非二进制字符串 (字符字符串)
tinytext, text, mediumtext, longtext
text 在定义时, 不需要定义长度, 也不会计算总长度。
text 类型在定义时, 不可给default值
-- c. binary, varbinary -----
类似于char和varchar, 用于保存二进制字符串, 也就是保存字节字符串而非字符字符串。
char, varchar, text 对应 binary, varbinary, blob.

3. 日期时间类型

一般用整型保存时间戳, 因为PHP可以很方便的将时间戳进行格式化。

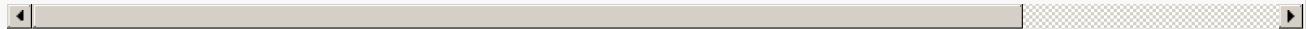
类型	字节数	描述	范围
datetime	8字节	日期及时间	1000-01-01 00:00:00 到 9999-12-31 23:59:59
date	3字节	日期	1000-01-01 到 9999-12-31
timestamp	4字节	时间戳	19700101000000 到 2038-01-19 03:14:07
time	3字节	时间	-838:59:59 到 838:59:59
year	1字节	年份	1901 - 2155

datetime YYYY-MM-DD hh:mm:ss
timestamp YY-MM-DD hh:mm:ss
YYYYMMDDhhmmss
YYMMDDhhmmss
YYYYYYMMDDhhmmss
YYMMDDhhmmss
date YYYY-MM-DD
YY-MM-DD
YYYYMMDD
YYMMDD
YYYYYYMMDD
YYMMDD
time hh:mm:ss
hhmmss
hhmmss
year YYYY
YY
YYYY
YY

4. 枚举和集合

-- 枚举(enum) -----
enum(val1, val2, val3...)
在已知的值中进行单选。最大数量为65535。
枚举值在保存时, 以2个字节的整型(smallint)保存。每个枚举值, 按保存的位置顺序, 从1开始逐一递增。
表现为字符串类型, 存储却是整型。
null 值的表示是null。

```
99 NULL值的索引是NULL。
100 空字符串错误值的索引值是0。
101
102 -- 集合 (set) -----
103 set(val1, val2, val3...)
104
105     create table tab ( gender set('男', '女', '无') );
106     insert into tab values ('男, 女');
107
108     最多可以有64个不同的成员。以bigint存储，共8个字节。采取位运算的形式。
109
110     当创建表时，SET成员值的尾部空格将自动被删除。
```



Tips：微信公众号Java后端，每日推送技术博文，欢迎关注。

选择类型

```
1 -- PHP角度
2 1. 功能满足
3 2. 存储空间尽量小，处理效率更高
4 3. 考虑兼容问题
5
6 -- IP存储 -----
7 1. 只需存储，可用字符串
8 2. 如果需计算，查找等，可存储为4个字节的无符号int，即unsigned
9     1) PHP函数转换
10        ip2long可转换为整型，但会出现携带符号问题。需格式化为无符号的整型。
11        利用sprintf函数格式化字符串
12        sprintf("%u", ip2long('192.168.3.134'));
13        然后用long2ip将整型转回IP字符串
14     2) MySQL函数转换(无符号整型, UNSIGNED)
15        INET_ATON('127.0.0.1') 将IP转为整型
16        INET_NTOA(2130706433) 将整型转为IP
```

列属性（列约束）

```
1 1. PRIMARY 主键
2     - 能唯一标识记录的字段，可以作为主键。
3     - 一个表只能有一个主键。
4     - 主键具有唯一性。
5     - 声明字段时，用 primary key 标识。
6         也可以在字段列表之后声明
7             例: create table tab ( id int, stu varchar(10), primary key (id));
8         - 主键字段的值不能为null。
9         - 主键可以由多个字段共同组成。此时需要在字段列表后声明的方法。
10            例: create table tab ( id int, stu varchar(10), age int, primary key (stu, age));
11 2. UNIQUE 唯一索引（唯一约束）
12     使得某字段的值也不能重复。
13 3. NULL 约束
14     null不是数据类型，是列的一个属性。
15     表示当前列是否可以为null，表示什么都没有。
16     null, 允许为空。默认。
17     not null, 不允许为空。
18     insert into tab values (null, 'val');
19         -- 此时表示将第一个字段的值设为null，取决于该字段是否允许为null
20 4. DEFAULT 默认值属性
21     当前字段的默认值。
```

```

22 insert into tab values (default, 'val');      -- 此时表示强制使用默认值。
23 create table tab ( add_time timestamp default current_timestamp );
24     -- 表示将当前时间的时间戳设为默认值。
25     current_date, current_time
26
27 5. AUTO_INCREMENT 自动增长约束
28     自动增长必须为索引（主键或unique）
29     只能存在一个字段为自动增长。
30     默认为1开始自动增长。可以通过表属性 auto_increment = x进行设置，或 alter table tbl auto_increment = x;
31
32 6. COMMENT 注释
33     例: create table tab ( id int ) comment '注释内容';
34
35 7. FOREIGN KEY 外键约束
36     用于限制主表与从表数据完整性。
37
38     alter table t1 add constraint `t1_t2_fk` foreign key (t1_id) references t2(id);
39     -- 将表t1的t1_id外键关联到表t2的id字段。
40     -- 每个外键都有一个名字，可以通过 constraint 指定
41
42     存在外键的表，称之为从表（子表），外键指向的表，称之为主表（父表）。
43     作用：保持数据一致性，完整性，主要目的是控制存储在外键表（从表）中的数据。
44
45     MySQL中，可以对InnoDB引擎使用外键约束：
46
47     语法：
48
49     foreign key (外键字段) references 主表名 (关联字段) [主表记录删除时的动作] [主表记录更新时的动作]
50
51     此时需要检测一个从表的外键需要约束为主表的已存在的值。外键在没有关联的情况下，可以设置为null.前提是该外键列，没有not null。
52
53     可以不指定主表记录更改或更新时的动作，那么此时主表的操作被拒绝。
54
55     如果指定了 on update 或 on delete: 在删除或更新时，有如下几个操作可以选择：
56
57     1. cascade, 级联操作。主表数据被更新（主键值更新），从表也被更新（外键值更新）。主表记录被删除，从表相关记录也被删除。
58
59     2. set null, 设置为null。主表数据被更新（主键值更新），从表的外键被设置为null。主表记录被删除，从表相关记录外键被设置成null。但
60
61     3. restrict, 拒绝父表删除和更新。
62
63     注意，外键只被InnoDB存储引擎所支持。其他引擎是不支持的。

```



建表规范

```

1   -- Normal Format, NF
2       - 每个表保存一个实体信息
3       - 每个具有一个ID字段作为主键
4       - ID主键 + 原子表
5
6   -- 1NF, 第一范式
7       字段不能再分，就满足第一范式。
8
9   -- 2NF, 第二范式
10      满足第一范式的前提下，不能出现部分依赖。
11      消除符合主键就可以避免部分依赖。增加单列关键字。
12
13   -- 3NF, 第三范式
14      满足第二范式的前提下，不能出现传递依赖。

```

```

1 SELECT [ALL|DISTINCT] select_expr FROM -> WHERE -> GROUP BY [合计函数] -> HAVING -> ORDER BY -> LIMIT
2 a. select_expr
3     -- 可以用 * 表示所有字段。
4         select * from tb;
5     -- 可以使用表达式（计算公式、函数调用、字段也是个表达式）
6         select stu, 29+25, now() from tb;
7     -- 可以为每个列使用别名。适用于简化列标识，避免多个列标识符重复。

```

使用 _ 为关键字，也可省略 _。

```
8      - 使用 as 关键字，也可省略 as。
9
10     select stu+10 as add10 from tb;
11
12 b. FROM 子句
13     用于标识查询来源。
14     -- 可以为表起别名。使用as关键字。
15
16     SELECT * FROM tb1 AS tt, tb2 AS bb;
17
18     -- from子句后，可以同时出现多个表。
19     -- 多个表会横向叠加到一起，而数据会形成一个笛卡尔积。
20
21     SELECT * FROM tb1, tb2;
22
23     -- 向优化符提示如何选择索引
24
25     USE INDEX、IGNORE INDEX、FORCE INDEX
26
27     SELECT * FROM table1 USE INDEX (key1,key2) WHERE key1=1 AND key2=2 AND key3=3;
28
29     SELECT * FROM table1 IGNORE INDEX (key3) WHERE key1=1 AND key2=2 AND key3=3;
30
31 c. WHERE 子句
32     -- 从from获得的数据源中进行筛选。
33     -- 整型1表示真，0表示假。
34     -- 表达式由运算符和运算数组成。
35     -- 运算数：变量（字段）、值、函数返回值
36     -- 运算符：
37
38         =, <=, >, !=, <, >=, >, !, &&, ||
39
40         in (not) null, (not) like, (not) in, (not) between and, is (not), and, or, not, xor
41
42         is/is not 加上ture/false/unknown, 检验某个值的真假
43
44         <=>与<>功能相同, <=>可用于null比较
45
46 d. GROUP BY 子句, 分组子句
47
48     GROUP BY 字段/别名 [排序方式]
49
50     分组后会进行排序。升序：ASC，降序：DESC
51
52     以下[合计函数]需配合 GROUP BY 使用：
53
54     count 返回不同的非NULL值数目 count(*)、count(字段)
55
56     sum 求和
57
58     max 求最大值
59
60     min 求最小值
61
62     avg 求平均值
63
64     group_concat 返回带有来自一个组的连接的非NULL值的字符串结果。组内字符串连接。
65
66 e. HAVING 子句, 条件子句
67
68     与 where 功能、用法相同，执行时机不同。
69
70     where 在开始时执行检测数据，对原数据进行过滤。
71
72     having 对筛选出的结果再次进行过滤。
73
74     having 字段必须是查询出来的，where 字段必须是数据表存在的。
75
76     where 不可以使用字段的别名，having 可以。因为执行WHERE代码时，可能尚未确定列值。
77
78     where 不可以使用合计函数。一般需用合计函数才会用 having
79
80     SQL标准要求HAVING必须引用GROUP BY子句中的列或用于合计函数中的列。
81
82 f. ORDER BY 子句, 排序子句
83
84     order by 排序字段/别名 排序方式 [,排序字段/别名 排序方式]...
85
86     升序：ASC，降序：DESC
87
88     支持多个字段的排序。
89
90 g. LIMIT 子句, 限制结果数量子句
91
92     仅对处理好的结果进行数量限制。将处理好的结果的看作是一个集合，按照记录出现的顺序，索引从0开始。
93
94     limit 起始位置，获取条数
95
96     省略第一个参数，表示从索引0开始。limit 获取条数
97
98 h. DISTINCT, ALL 选项
99
100    distinct 去除重复记录
101
102    默认为 all，全部记录
```

```

1 将多个select查询的结果组合成一个结果集合。
2 SELECT ... UNION [ALL|DISTINCT] SELECT ...
3 默认 DISTINCT 方式，即所有返回的行都是唯一的
4 建议，对每个SELECT查询加上小括号包裹。
5 ORDER BY 排序时，需加上 LIMIT 进行结合。
6 需要各select查询的字段数量一样。
7 每个select查询的字段列表(数量、类型)应一致，因为结果中的字段名以第一条select语句为准。

```

子查询

```

1 - 子查询需用括号包裹。
2 -- from型
3   from后要求是一个表，必须给子查询结果取个别名。
4 - 简化每个查询内的条件。
5   - from型需将结果生成一个临时表格，可用以原表的锁定的释放。
6   - 子查询返回一个表，表型子查询。
7     select * from (select * from tb where id>0) as subfrom where id>1;
8 -- where型
9   - 子查询返回一个值，标量子查询。
10  - 不需要给子查询取别名。
11  - where子查询内的表，不能直接用以更新。
12  select * from tb where money = (select max(money) from tb);
13 -- 列子查询
14   如果子查询结果返回的是一列。
15   使用 in 或 not in 完成查询
16   exists 和 not exists 条件
17   如果子查询返回数据，则返回1或0。常用于判断条件。
18   select column1 from t1 where exists (select * from t2);
19 -- 行子查询
20   查询条件是一个行。
21   select * from t1 where (id, gender) in (select id, gender from t2);
22   行构造符: (col1, col2, ...) 或 ROW(col1, col2, ...)
23   行构造符通常用于与对能返回两个或两个以上列的子查询进行比较。
24 -- 特殊运算符
25   != all()    相当于 not in
26   = some()    相当于 in。any 是 some 的别名
27   != some()   不等同于 not in，不等于其中某一个。
28   all, some 可以配合其他运算符一起使用。

```

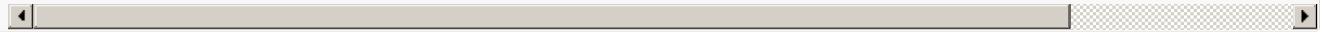
连接查询(join)

```

1 将多个表的字段进行连接，可以指定连接条件。
2 -- 内连接(inner join)
3   - 默认就是内连接，可省略inner。
4   - 只有数据存在时才能发送连接。即连接结果不能出现空行。
5   on 表示连接条件。其条件表达式与where类似。也可以省略条件（表示条件永远为真）
6   也可用where表示连接条件。
7   还有 using，但需字段名相同。 using(字段名)
8   -- 交叉连接 cross join
9     即，没有条件的内连接。
      select * from tb1 cross join tb2;

```

```
10 -- 外连接(outer join)
11   - 如果数据不存在，也会出现在连接结果中。
12   -- 左外连接 left join
13     如果数据不存在，左表记录会出现，而右表为null填充
14   -- 右外连接 right join
15     如果数据不存在，右表记录会出现，而左表为null填充
16 -- 自然连接(natural join)
17   自动判断连接条件完成连接。
18   相当于省略了using，会自动查找相同字段名。
19   natural join
20   natural left join
21   natural right join
22
23 select info.id, info.name, info.stu_num, extra_info.hobby, extra_info.sex from info, extra_info where info.stu_num
24
```



导出

```
1 select * into outfile 文件地址 [控制格式] from 表名; -- 导出表数据
2
3 load data [local] infile 文件地址 [replace|ignore] into table 表名 [控制格式]; -- 导入数据
4   生成的数据默认的分隔符是制表符
5   local未指定，则数据文件必须在服务器上
6   replace 和 ignore 关键词控制对现有的唯一键记录的重复的处理
7 -- 控制格式
8 fields 控制字段格式
9 默认: fields terminated by '\t' enclosed by '' escaped by '\\'
10    terminated by 'string' -- 终止
11    enclosed by 'char' -- 包裹
12    escaped by 'char' -- 转义
13 -- 示例:
14   SELECT a,b,a+b INTO OUTFILE '/tmp/result.text'
15     FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY ''
16     LINES TERMINATED BY '\n'
17     FROM test_table;
18 lines 控制行格式
19 默认: lines terminated by '\n'
20    terminated by 'string' -- 终止
```

INSERT

```
1 select语句获得的数据可以用insert插入。
2 可以省略对列的指定，要求 values () 括号内，提供给了按照列顺序出现的所有字段的值。
3   或者使用set语法。
4     INSERT INTO tbl_name SET field=value,...;
5 可以一次性使用多个值，采用(),(),()等形式。
6     INSERT INTO tbl_name VALUES (),(),();
7 可以在列值指定时，使用表达式。
8     INSERT INTO tbl_name VALUES (field_value, 10+10, now());
9 可以使用一个特殊值 DEFAULT，表示该列使用默认值。
10    INSERT INTO tbl_name VALUES (field_value, DEFAULT);
11 可以通过一个查询的结果，作为需要插入的值。
12    INSERT INTO tbl_name SELECT ...;
```

可以指定在插入的值出现主键（或唯一索引）冲突时，更新其他非主键列的信息

```
13 可以指定往插入的值出现主键(或唯一索引)冲突时, 更新其他非主键列的信息。
14 INSERT INTO tbl_name VALUES/SET/SELECT ON DUPLICATE KEY UPDATE 字段=值, ...;
```

DELETE

```
1 DELETE FROM tbl_name [WHERE where_definition] [ORDER BY ...] [LIMIT row_count]
2 按照条件删除。where
3 指定删除的最多记录数。limit
4 可以通过排序条件删除。order by + limit
5 支持多表删除, 使用类似连接语法。
6 delete from 需要删除数据多表1, 表2 using 表连接操作 条件。
```

TRUNCATE

```
1 TRUNCATE [TABLE] tbl_name
2 清空数据
3 删除重建表
4 区别:
5 1, truncate 是删除表再创建, delete 是逐条删除
6 2, truncate 重置auto_increment的值。而delete不会
7 3, truncate 不知道删除了几条, 而delete知道。
8 4, 当被用于带分区的表时, truncate 会保留分区
```

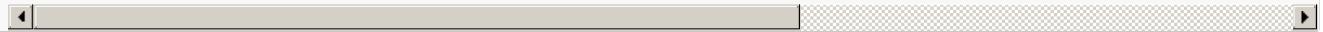
备份与还原

```
1 备份, 将数据的结构与表内数据保存起来。
2 利用 mysqldump 指令完成。
3 -- 导出
4 mysqldump [options] db_name [tables]
5 mysqldump [options] --database DB1 [DB2 DB3...]
6 mysqldump [options] --all-database
7 1. 导出一张表
8     mysqldump -u用户名 -p密码 库名 表名 > 文件名(D:/a.sql)
9 2. 导出多张表
10    mysqldump -u用户名 -p密码 库名 表1 表2 表3 > 文件名(D:/a.sql)
11 3. 导出所有表
12    mysqldump -u用户名 -p密码 库名 > 文件名(D:/a.sql)
13 4. 导出一个库
14    mysqldump -u用户名 -p密码 --lock-all-tables --database 库名 > 文件名(D:/a.sql)
15 可以 -w携带WHERE条件
16 -- 导入
17 1. 在登录mysql的情况下:
18     source 备份文件
19 2. 在不登录的情况下
20     mysql -u用户名 -p密码 库名 < 备份文件
```

视图

```
1 什么是视图:
2   视图是一个虚拟表, 其内容由查询定义。同真实的表一样, 视图包含一系列带有名称的列和行数据。但是, 视图并不在数据库中以存储的数据值集形式
3   存储, 视图具有表结构文件, 但不存在数据文件。
4   对其中所引用的基础表来说, 视图的作用类似于筛选。定义视图的筛选可以来自当前或其它数据库的一个或多个表, 或者其它视图。通过视图进行查询
```

5 视图是存储在数据库中的查询的sql语句，它主要出于两种原因：安全原因，视图可以隐藏一些数据，如：社会保险基金表，可以用视图只显示姓名，
6 -- 创建视图
7 CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW view_name [(column_list)] AS select_stateme
8 - 视图名必须唯一，同时不能与表重名。
9 - 视图可以使用select语句查询到的列名，也可以自己指定相应的列名。
10 - 可以指定视图执行的算法，通过ALGORITHM指定。
11 - column_list如果存在，则数目必须等于SELECT语句检索的列数
12 -- 查看结构
13 SHOW CREATE VIEW view_name
14 -- 删除视图
15 - 删除视图后，数据依然存在。
16 - 可同时删除多个视图。
17 DROP VIEW [IF EXISTS] view_name ...
18 -- 修改视图结构
19 - 一般不修改视图，因为不是所有的更新视图都会映射到表上。
20 ALTER VIEW view_name [(column_list)] AS select_statement
21 -- 视图作用
22 1. 简化业务逻辑
23 2. 对客户端隐藏真实的表结构
24 -- 视图算法(ALGORITHM)
25 MERGE 合并
26 将视图的查询语句，与外部查询需要先合并再执行！
27 TEMPTABLE 临时表
28 将视图执行完毕后，形成临时表，再做外层查询！
29 UNDEFINED 未定义(默认)，指的是MySQL自主去选择相应的算法。



事务(transaction)

1 事务是指逻辑上的一组操作，组成这组操作的各个单元，要不全成功要不全失败。
2 - 支持连续SQL的集体成功或集体撤销。
3 - 事务是数据库在数据完整性方面的一个功能。
4 - 需要利用 InnoDB 或 BDB 存储引擎，对自动提交的特性支持完成。
5 - InnoDB被称为事务安全型引擎。
6 -- 事务开启
7 START TRANSACTION; 或者 BEGIN;
8 开启事务后，所有被执行的SQL语句均被认作当前事务内的SQL语句。
9 -- 事务提交
10 COMMIT;
11 -- 事务回滚
12 ROLLBACK;
13 如果部分操作发生问题，映射到事务开启前。
14 -- 事务的特性
15 1. 原子性 (Atomicity)
16 事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
17 2. 一致性 (Consistency)
18 事务前后数据的完整性必须保持一致。
19 - 事务开始和结束时，外部数据一致
20 - 在整个事务过程中，操作是连续的
21 3. 隔离性 (Isolation)
22 多个用户并发访问数据库时，一个用户的事务不能被其它用户的事物所干扰，多个并发事务之间的数据要相互隔离。
23 4. 持久性 (Durability)
24 一个事务一旦被提交，它对数据库中的数据改变就是永久性的。
25 -- 事务的实现
1 要求是事务支持的类型

```
1. 要不是事务支持的表类型
2. 执行一组相关操作前开启事务
3. 整组操作完成后，都成功，则提交；如果存在失败，选择回滚，则会回到事务开始的备份点。
-- 事务的原理
利用InnoDB的自动提交(autocommit)特性完成。
普通的MySQL执行语句后，当前的数据提交操作均可被其他客户端可见。
而事务是暂时关闭“自动提交”机制，需要commit提交持久化数据操作。
-- 注意
1. 数据定义语言（DDL）语句不能被回滚，比如创建或取消数据库的语句，和创建、取消或更改表或存储的子程序的语句。
2. 事务不能被嵌套
-- 保存点
SAVEPOINT 保存点名称 -- 设置一个事务保存点
ROLLBACK TO SAVEPOINT 保存点名称 -- 回滚到保存点
RELEASE SAVEPOINT 保存点名称 -- 删除保存点
-- InnoDB自动提交特性设置
SET autocommit = 0|1; 0表示关闭自动提交，1表示开启自动提交。
- 如果关闭了，那普通操作的结果对其他客户端也不可见，需要commit提交后才能持久化数据操作。
- 也可以关闭自动提交来开启事务。但与START TRANSACTION不同的是，
    SET autocommit是永久改变服务器的设置，直到下次再次修改该设置。（针对当前连接）
    而START TRANSACTION记录开启前的状态，而一旦事务提交或回滚后就需要再次开启事务。（针对当前事务）

```

锁表

```
1 表锁定只用于防止其它客户端进行不正当地读取和写入
2 MyISAM 支持表锁，InnoDB 支持行锁
-- 锁定
3 LOCK TABLES tbl_name [AS alias]
-- 解锁
4 UNLOCK TABLES
```

触发器

```
1 触发程序是与表有关的命名数据库对象，当该表出现特定事件时，将激活该对象
2 监听：记录的增加、修改、删除。
-- 创建触发器
3 CREATE TRIGGER trigger_name trigger_time trigger_event ON tbl_name FOR EACH ROW trigger_stmt
参数：
4 trigger_time是触发程序的动作时间。它可以是 before 或 after，以指明触发程序是在激活它的语句之前或之后触发。
5 trigger_event指明了激活触发程序的语句的类型
6 INSERT：将新行插入表时激活触发程序
7 UPDATE：更改某一行时激活触发程序
8 DELETE：从表中删除某一行时激活触发程序
9 tbl_name：监听的表，必须是永久性的表，不能将触发程序与TEMPORARY表或视图关联起来。
10 trigger_stmt：当触发程序激活时执行的语句。执行多个语句，可使用BEGIN...END复合语句结构
-- 删除
11 DROP TRIGGER [schema_name.]trigger_name
12 可以使用old和new代替旧的和新的数据
13 更新操作，更新前是old，更新后是new。
14 删操作，只有old。
15 增加操作，只有new。
-- 注意
16 1. 对于具有相同触发程序动作时间和事件的给定表，不能有两个触发程序。
```

```

1 -- 字符连接函数
2 concat(str1,str2,...])
3 concat_ws(separator,str1,str2,...)
4
5 -- 分支语句
6 if 条件 then
7     执行语句
8 elseif 条件 then
9     执行语句
10 else
11     执行语句
12 end if;
13
14 -- 修改最外层语句结束符
15 delimiter 自定义结束符号
16     SQL语句
17 自定义结束符号
18 delimiter ;      -- 修改回原来的分号
19
20 -- 语句块包裹
21 begin
22     语句块
23 end
24
25 -- 特殊的执行
26 1. 只要添加记录，就会触发程序。
27 2. Insert into on duplicate key update 语法会触发：
28     如果没有重复记录，会触发 before insert, after insert;
29     如果有重复记录并更新，会触发 before insert, before update, after update;
30     如果有重复记录但是没有发生更新，则触发 before insert, before update
31 3. Replace 语法 如果有记录，则执行 before insert, before delete, after delete, after insert

```

SQL编程

```

1 --// 局部变量 -----
2 -- 变量声明
3 declare var_name[,...] type [default value]
4 这个语句被用来声明局部变量。要给变量提供一个默认值，请包含一个default子句。值可以被指定为一个表达式，不需要为一个常数。如果没有de
5 -- 赋值
6 使用 set 和 select into 语句为变量赋值。
7 - 注意：在函数内是可以使用全局变量（用户自定义的变量）
8
9
10 --// 全局变量 -----
11 -- 定义、赋值
12 set 语句可以定义并为变量赋值。
13 set @var = value;
14 也可以使用select into语句为变量初始化并赋值。这样要求select语句只能返回一行，但是可以是多个字段，就意味着同时为多个变量进行赋值，变
15 还可以把赋值语句看作一个表达式，通过select执行完成。此时为了避免=被当作关系运算符看待，使用:=代替。（set语句可以使用= 和 :=）。
16 select @var:=20;
17 select @v1:=id, @v2=name from t1 limit 1;
18 select * from tbl_name where @var:=30;
    select into 可以将表中查询获得的数据赋给变量。

```

```
19    -| select max(height) into @max_height from tb;
20 -- 自定义变量名
21 为了避免select语句中，用户自定义的变量与系统标识符（通常是字段名）冲突，用户自定义变量在变量名前使用@作为开始符号。
22 @var=10;
23     - 变量被定义后，在整个会话周期都有效（登录到退出）
24
25
26
27 ---// 控制结构 -----
28 -- if语句
29 if search_condition then
30     statement_list
31 [elseif search_condition then
32     statement_list]
33 ...
34 [else
35     statement_list]
36 end if;
37 -- case语句
38 CASE value WHEN [compare-value] THEN result
39 [WHEN [compare-value] THEN result ...]
40 [ELSE result]
41 END
42 -- while循环
43 [begin_label:] while search_condition do
44     statement_list
45 end while [end_label];
46 - 如果需要在循环内提前终止 while循环，则需要使用标签；标签需要成对出现。
47     -- 退出循环
48         退出整个循环 leave
49         退出当前循环 iterate
50     通过退出的标签决定退出哪个循环
51
52
53
54 ---// 内置函数 -----
55 -- 数值函数
56 abs(x)          -- 绝对值 abs(-10.9) = 10
57 format(x, d)   -- 格式化千分位数值 format(1234567.456, 2) = 1,234,567.46
58 ceil(x)         -- 向上取整 ceil(10.1) = 11
59 floor(x)        -- 向下取整 floor (10.1) = 10
60 round(x)        -- 四舍五入去整
61 mod(m, n)       -- m%n m mod n 求余 10%3=1
62 pi()            -- 获得圆周率
63 pow(m, n)       -- m^n
64 sqrt(x)         -- 算术平方根
65 rand()           -- 随机数
66 truncate(x, d) -- 截取d位小数
67 -- 时间日期函数
68 now(), current_timestamp();      -- 当前日期时间
69 current_date();                  -- 当前日期
70 current_time();                  -- 当前时间
71 date('yyyy-mm-dd hh:ii:ss');    -- 获取日期部分
72 time('yyyy-mm-dd hh:ii:ss');    -- 获取时间部分
```

```
73 date_format('yyyy-mm-dd hh:ii:ss', '%d %y %a %d %m %b %j'); -- 格式化时间
74 unix_timestamp(); -- 获得unix时间戳
75 from_unixtime(); -- 从时间戳获得时间
76 -- 字符串函数
77 length(string) -- string长度, 字节
78 char_length(string) -- string的字符个数
79 substring(str, position [,length]) -- 从str的position开始, 取length个字符
80 replace(str ,search_str ,replace_str) -- 在str中用replace_str替换search_str
81 instr(string ,substring) -- 返回substring首次在string中出现的位置
82 concat(string [,...]) -- 连接字符串
83 charset(str) -- 返回字符串字符集
84 lcase(string) -- 转换成小写
85 left(string, length) -- 从string2中的左边起取length个字符
86 load_file(file_name) -- 从文件读取内容
87 locate(substring, string [,start_position]) -- 同instr, 但可指定开始位置
88 lpad(string, length, pad) -- 重复用pad加在string开头, 直到字符串长度为length
89 ltrim(string) -- 去除前端空格
90 repeat(string, count) -- 重复count次
91 rpad(string, length, pad) -- 在str后用pad补充, 直到长度为length
92 rtrim(string) -- 去除后端空格
93 strcmp(string1 ,string2) -- 逐字符比较两字符串大小
94 -- 流程函数
95 case when [condition] then result [when [condition] then result ...] [else result] end 多分支
96 if(expr1,expr2,expr3) 双分支。
97 -- 聚合函数
98 count()
99 sum();
100 max();
101 min();
102 avg();
103 group_concat()
104 -- 其他常用函数
105 md5();
106 default();
107
108 --// 存储函数, 自定义函数 -----
109 -- 新建
110 CREATE FUNCTION function_name (参数列表) RETURNS 返回值类型
111     函数体
112     - 函数名, 应该合法的标识符, 并且不应该与已有的关键字冲突。
113     - 一个函数应该属于某个数据库, 可以使用db_name.funciton_name的形式执行当前函数所属数据库, 否则为当前数据库。
114     - 参数部分, 由"参数名"和"参数类型"组成。多个参数用逗号隔开。
115     - 函数体由多条可用的mysql语句, 流程控制, 变量声明等语句构成。
116     - 多条语句应该使用 begin...end 语句块包含。
117     - 一定要有 return 返回值语句。
118 -- 删除
119     DROP FUNCTION [IF EXISTS] function_name;
120 -- 查看
121     SHOW FUNCTION STATUS LIKE 'partten'
122     SHOW CREATE FUNCTION function_name;
123 -- 修改
124     ALTER FUNCTION function_name 函数选项
125
126 --// 存储过程, 自定义功能 -----
```

```
127 -- 定义
128 存储存储过程 是一段代码（过程），存储在数据库中的sql组成。
129 一个存储过程通常用于完成一段业务逻辑，例如报名，交班费，订单入库等。
130 而一个函数通常专注与某个功能，视为其他程序服务的，需要在其他语句中调用函数才可以，而存储过程不能被其他调用，是自己执行 通过call执行。
131 -- 创建
132 CREATE PROCEDURE sp_name (参数列表)
133     过程体
134 参数列表：不同于函数的参数列表，需要指明参数类型
135 IN, 表示输入型
136 OUT, 表示输出型
137 INOUT, 表示混合型
138 注意，没有返回值。
139
140
141 /* 存储过程 */
142 存储过程是一段可执行性代码的集合。相比函数，更偏向于业务逻辑。
143 调用：CALL 过程名
144 -- 注意
145 - 没有返回值。
146 - 只能单独调用，不可夹杂在其他语句中
147 -- 参数
148 IN|OUT|INOUT 参数名 数据类型
149 IN      输入：在调用过程中，将数据输入到过程体内部的参数
150 OUT     输出：在调用过程中，将过程体处理完的结果返回到客户端
151 INOUT   输入输出：既可输入，也可输出
152 -- 语法
153 CREATE PROCEDURE 过程名 (参数列表)
154 BEGIN
155     过程体
156 END
157
```

用户和权限管理

```
1 -- root密码重置
2 1. 停止MySQL服务
3 2. [Linux] /usr/local/mysql/bin/safe_mysqld --skip-grant-tables &
4     [Windows] mysqld --skip-grant-tables
5 3. use mysql;
6 4. UPDATE `user` SET PASSWORD=PASSWORD("密码") WHERE `user` = "root";
7 5. FLUSH PRIVILEGES;
8 用户信息表：mysql.user
9 -- 刷新权限
10 FLUSH PRIVILEGES;
11 -- 增加用户
12 CREATE USER 用户名 IDENTIFIED BY [PASSWORD] 密码(字符串)
13     - 必须拥有mysql数据库的全局CREATE USER权限，或拥有INSERT权限。
14     - 只能创建用户，不能赋予权限。
15     - 用户名，注意引号：如 'user_name'@'192.168.1.1'
16     - 密码也需引号，纯数字密码也要加引号
17     - 要在纯文本中指定密码，需忽略PASSWORD关键词。要把密码指定为由PASSWORD()函数返回的混编值，需包含关键字PASSWORD
18 -- 重命名用户
19 RENAME USER old_user TO new_user
20 -- 设置密码
```

```
21 SET PASSWORD = PASSWORD('密码') -- 为当前用户设置密码
22 SET PASSWORD FOR 用户名 = PASSWORD('密码') -- 为指定用户设置密码
23 -- 删除用户
24 DROP USER 用户名
25 -- 分配权限/添加用户
26 GRANT 权限列表 ON 表名 TO 用户名 [IDENTIFIED BY [PASSWORD] 'password']
27     - all privileges 表示所有权限
28     - *.* 表示所有库的所有表
29     - 库名.表名 表示某库下面的某表
30     GRANT ALL PRIVILEGES ON `pms`.* TO 'pms'@'%' IDENTIFIED BY 'pms0817';
31 -- 查看权限
32 SHOW GRANTS FOR 用户名
33     -- 查看当前用户权限
34     SHOW GRANTS; 或 SHOW GRANTS FOR CURRENT_USER; 或 SHOW GRANTS FOR CURRENT_USER();
35 -- 撤消权限
36 REVOKE 权限列表 ON 表名 FROM 用户名
37 REVOKE ALL PRIVILEGES, GRANT OPTION FROM 用户名 -- 撤销所有权限
38 -- 权限层级
39 -- 要使用GRANT或REVOKE，您必须拥有GRANT OPTION权限，并且您必须用于您正在授予或撤销的权限。
40 全局层级：全局权限适用于一个给定服务器中的所有数据库，mysql.user
41     GRANT ALL ON *.*和REVOKE ALL ON *.*只授予和撤销全局权限。
42 数据库层级：数据库权限适用于一个给定数据库中的所有目标，mysql.db, mysql.host
43     GRANT ALL ON db_name.*和REVOKE ALL ON db_name.*只授予和撤销数据库权限。
44 表层级：表权限适用于一个给定表中的所有列，mysql.talbes_priv
45     GRANT ALL ON db_name.tbl_name和REVOKE ALL ON db_name.tbl_name只授予和撤销表权限。
46 列层级：列权限适用于一个给定表中的单一列，mysql.columns_priv
47     当使用REVOKE时，您必须指定与被授权列相同的列。
48 -- 权限列表
49 ALL [PRIVILEGES] -- 设置除GRANT OPTION之外的所有简单权限
50 ALTER -- 允许使用ALTER TABLE
51 ALTER ROUTINE -- 更改或取消已存储的子程序
52 CREATE -- 允许使用CREATE TABLE
53 CREATE ROUTINE -- 创建已存储的子程序
54 CREATE TEMPORARY TABLES -- 允许使用CREATE TEMPORARY TABLE
55 CREATE USER -- 允许使用CREATE USER, DROP USER, RENAME USER和REVOKE ALL PRIVILEGES。
56 CREATE VIEW -- 允许使用CREATE VIEW
57 DELETE -- 允许使用DELETE
58 DROP -- 允许使用DROP TABLE
59 EXECUTE -- 允许用户运行已存储的子程序
60 FILE -- 允许使用SELECT...INTO OUTFILE和LOAD DATA INFILE
61 INDEX -- 允许使用CREATE INDEX和DROP INDEX
62 INSERT -- 允许使用INSERT
63 LOCK TABLES -- 允许对您拥有SELECT权限的表使用LOCK TABLES
64 PROCESS -- 允许使用SHOW FULL PROCESSLIST
65 REFERENCES -- 未被实施
66 RELOAD -- 允许使用FLUSH
67 REPLICATION CLIENT -- 允许用户询问从属服务器或主服务器的地址
68 REPLICATION SLAVE -- 用于复制型从属服务器（从主服务器中读取二进制日志事件）
69 SELECT -- 允许使用SELECT
70 SHOW DATABASES -- 显示所有数据库
71 SHOW VIEW -- 允许使用SHOW CREATE VIEW
72 SHUTDOWN -- 允许使用mysqladmin shutdown
73 SUPER -- 允许使用CHANGE MASTER, KILL, PURGE MASTER LOGS和SET GLOBAL语句, mysqladmin debug命令；允许您连接（一次），即使您
    UPDATE -- 允许使用UPDATE
```

```
74 USAGE -- “无权限”的同义词
75 GRANT OPTION -- 允许授予权限
76
```

表维护

```
1 -- 分析和存储表的关键字分布
2 ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE 表名 ...
3 -- 检查一个或多个表是否有错误
4 CHECK TABLE tbl_name [, tbl_name] ... [option] ...
5 option = {QUICK | FAST | MEDIUM | EXTENDED | CHANGED}
6 -- 整理数据文件的碎片
7 OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [, tbl_name] ...
```

杂项

```
1 1. 可用反引号 (`) 为标识符 (库名、表名、字段名、索引、别名) 包裹, 以避免与关键字重名! 中文也可以作为标识符!
2 2. 每个库目录存在一个保存当前数据库的选项文件db.opt。
3 3. 注释:
4     单行注释 # 注释内容
5     多行注释 /* 注释内容 */
6     单行注释 -- 注释内容      (标准SQL注释风格, 要求双破折号后加一空格符 (空格、TAB、换行等) )
7 4. 模式通配符:
8     _ 任意单个字符
9     % 任意多个字符, 甚至包括零字符
10    单引号需要进行转义 \''
11 5. CMD命令行内的语句结束符可以为 ";" , "\G" , "\g" , 仅影响显示结果。其他地方还是用分号结束。delimiter 可修改当前行
12 6. SQL对大小写不敏感
13 7. 清除已有语句: \c
```

作者 | 格物

链接 | shockerli.net/post/1000-line-mysql-note/

-END-

推荐阅读

1. [大数据开发经验架构师整理大数据学习路线](#)
2. [项目中常用到的 19 条 MySQL 优化](#)
3. [一文读懂并实操 Nginx 反向代理](#)
4. [丢掉 Postman ! 我选择了 IDEA REST Client](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章, 点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

一次非常有趣的 SQL 优化经历

风过无痕 Java后端 2019-12-07

点击上方 Java后端, 选择 **设为星标**

优质文章, 及时送达

作者 | 风过无痕

链接 | cnblogs.com/tangyanbo/p/4462734.html

场景

我用的数据库是mysql5.6, 下面简单的介绍下场景

课程表:

```
create table Course(
c_id int PRIMARY KEY,
name varchar(10)
)
```

数据100条

学生表:

```
create table Student(
id int PRIMARY KEY,
name varchar(10)
)
```

数据70000条

学生成绩表SC:

```
CREATE table SC(
sc_id int PRIMARY KEY,
s_id int,
c_id int,
score int
)
```

数据70w条

查询目的:

查找语文考100分的考生

查询语句:

```
select s.* from Student s where s.s_id in
(select s_id from SC sc where sc.c_id = 0 and sc.score = 100 )
```

晕，为什么这么慢，先来查看下查询计划：

EXPLAIN

```
select s.* from Student s where s.s_id in
(select s_id from SC sc where sc.c_id = 0 and sc.score = 100 )
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	s	ALL	(Null)	(Null)	(Null)	(Null)	70007	Using where
2	DEPENDENT SUBQUERY	sc	ALL	(Null)	(Null)	(Null)	(Null)	770011	Using where

发现没有用到索引，type全是ALL，那么首先想到的就是建立一个索引，建立索引的字段当然是在where条件的字段了。

先给sc表的c_id和score建个索引

```
CREATE index sc_c_id_index on SC(c_id);
CREATE index sc_score_index on SC(score);
```

再次执行上述查询语句，时间为: 1.054s

快了3w多倍，大大缩短了查询时间，看来索引能极大程度的提高查询效率，建索引很有必要，很多时候都忘记建索引了，数据量小的时候压根没感觉，这优化的感觉挺爽。

但是1s的时间还是太长了，还能进行优化吗，仔细看执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	s	ALL	(Null)	(Null)	(Null)	(Null)	70007	Using where
2	DEPENDENT SUBQUERY	sc	ref	sc_s_id_index,sc_score_index,sc_c_id_index	sc_score_index	5	const	8	Using where

查看优化后的sql:

```
SELECT
`YSB`.`s`.`s_id` AS `s_id`,
`YSB`.`s`.`name` AS `name`
FROM
`YSB`.`Student` `s`
WHERE
<in_optimizer>(
`YSB`.`s`.`s_id`,<EXISTS>(
SELECT
FROM
`YSB`.`SC` `sc`
WHERE
(
(`YSB`.`sc`.`c_id` = 0)
AND (`YSB`.`sc`.`score` = 100)
AND (
<CACHE>(`YSB`.`s`.`s_id`) = `YSB`.`sc`.`s_id`
)
)
)
```

怎么查看优化后的语句？

方法如下（在命令窗口执行）：

```
mysql> EXPLAIN EXTENDED
select s.* from Student s where s.s_id in (select s_id from SC sc where sc.score = 100 and sc.c_id = 0 )
;

mysql> show warnings;
+-----+-----+
| Level | Code | Message
+-----+-----+
| Note | 1003 | select `YSB`.`s`.`s_id` AS `s_id`, `1` from `YSB`.`SC` `sc` where ((`YSB`.`sc`.`c_id` = 0
+-----+-----+
1 row in set
```

有type=all

按照我之前的想法，该sql的执行的顺序应该是先执行子查询

```
select s_id from SC sc where sc.c_id = 0 and sc.score = 100
```

耗时：0.001s

得到如下结果：

s_id
7
29
4999

然后再执行

```
select s.* from Student s where s.s_id in(7,29,5000)
```

耗时：0.001s

这样就是相当快了啊，Mysql竟然不是先执行里层的查询，而是将sql优化成了exists子句，并出现了EPENDENT SUBQUERY，mysql是先执行外层查询，再执行里层的查询，这样就要循环70007*8次。

那么改用连接查询呢？

```
SELECT s.* from
Student s
INNER JOIN SC sc
on sc.s_id = s.s_id
where sc.c_id=0 and sc.score=100
```

这里为了重新分析连接查询的情况，先暂时删除索引sc_c_id_index, sc_score_index

执行时间是：0.057s

效率有所提高，看看执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶ 1	SIMPLE	sc	ALL	(Null)	(Null)	(Null)	(Null)	770011	Using where
1	SIMPLE	s	eq_ref	PRIMARY	PRIMAR 4		YSB.sc.s_id	1	

这里有连表的情况出现，我猜想是不是要给sc表的s_id建立个索引

```
CREATE index sc_s_id_index on SC(s_id);
show index from SC
```

Table	Non_unique	Key_name	Seq_in_index
▶ SC	0	PRIMARY	
SC	1	sc_s_id_index	

再执行连接查询

时间: 1.076s

竟然时间还变长了，什么原因？查看执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶ 1	SIMPLE	s	ALL	PRIMARY	(Null)	(Null)	(Null)	70007	
1	SIMPLE	sc	ref	sc_s_id_index	sc_s_id_index	5	YSB.s.s_id	11	Using where

优化后的查询语句为：

```
SELECT
`YSB`.`s`.`s_id` AS `s_id`,
`YSB`.`s`.`name` AS `name`
FROM
`YSB`.`Student` `s`
JOIN `YSB`.`SC` `sc`
WHERE
(
(
`YSB`.`sc`.`s_id` = `YSB`.`s`.`s_id`
)
AND (`YSB`.`sc`.`score` = 100)
AND (`YSB`.`sc`.`c_id` = 0)
)
```

貌似是先做的连接查询，再进行的where条件过滤

回到前面的执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶ 1	SIMPLE	sc	ALL	(Null)	(Null)	(Null)	(Null)	770011	Using where
1	SIMPLE	s	eq_ref	PRIMARY	PRIMAR 4		YSB.sc.s_id	1	

这里是先做的where条件过滤，再做连表，执行计划还不是固定的，那么我们先看下标准的sql执行顺序：

```

(8) SELECT <9> DISTINCT<select_list>
(1) FROM <left_table>
(3) <join_type>JOIN<right_table>
(2)      ON<join_condition>
(4) WHERE<where_condition>
(5) GROUP BY<group_by_list>
(6) WITH {CUBE|ROLLUP}
(7) HAVING<having_condition>
(10) ORDER BY<order_by_list>
(11) LIMIT <limit_number>

```

图 3-1 逻辑查询处理的步骤序号

正常情况下是先join再进行where过滤，但是我们这里的情况，如果先join，将会有70w条数据发送join，因此先执行where过滤是明智方案，现在为了排除mysql的查询优化，我自己写一条优化后的sql

```

SELECT
  s.*
FROM
(
  SELECT
    *
  FROM
    SC sc
  WHERE
    sc.c_id = 0
    AND sc.score = 100
) t
INNER JOIN Students s ON t.s_id = s.s_id

```

即先执行sc表的过滤，再进行表连接

执行时间为：0.054s

和之前没有建s_id索引的时间差不多

查看执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	(Null)	(Null)	(Null)	(Null)	3	
▶ 1	PRIMARY	s	eq_ref	PRIMARY	PRIMAR	4	t.s_id	1	
2	DERIVED	sc	ALL	(Null)	(Null)	(Null)	(Null)	770011	Using where

先提取sc再连表，这样效率就高多了，现在的问题是提取sc的时候出现了扫描表，那么现在可以明确需要建立相关索引

```

CREATE index sc_c_id_index on SC(c_id);
CREATE index sc_score_index on SC(score);

```

再执行查询：

```
SELECT
  s.*
FROM
(
  SELECT
    *
  FROM
    SC sc
  WHERE
    sc.c_id = 0
  AND sc.score = 100
) t
```

```
INNER JOIN Student s ON t.s_id = s.s_id
```

执行时间为：0.001s

这个时间相当靠谱，快了50倍

执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	(Null)	(Null)	(Null)	(Null)	3	
1	PRIMARY	s	eq_ref	PRIMARY	PRIMARY	4	t.s_id	1	
2	DERIVED	sc	ref	sc_c_id_index,sc_score_index	sc_score_inc	5		27	Using where

我们会看到，先提取sc，再连表，都用到了索引。

那么再来执行下sql：

```
SELECT s.* from
Student s
INNER JOIN SC sc
on sc.s_id = s.s_id
```

```
where sc.c_id=0 and sc.score=100
```

执行时间0.001s

执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	sc	ref	sc_c_id_index,sc_score_index	sc_score_index	5	const	27	100	Using where
1	SIMPLE	s	eq_ref	PRIMARY	PRIMARY	4	YSB.sc.s_id	1	100	

这里是mysql进行了查询语句优化，先执行了where过滤，再执行连接操作，且都用到了索引。

调整内容为SC表的数据增长到300W,学生分数更为离散。

先回顾下：

```
show index from SC
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Coll
SC	0	PRIMARY		sc_id	A
SC	1	sc_c_id_index	1	c_id	A
SC	1	sc_score_index	1	score	A

执行sql

```
SELECT s.* from
Student s
INNER JOIN SC sc
on sc.s_id = s.s_id
```

```
where sc.c_id=81 and sc.score=84
```

执行时间：0.061s

这个时间稍微慢了点

执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	sc	index_m	sc_c_id_index,sc_score_index	sc_score_index,sc_5,5	(Null)	959		Using intersect(sc_score_index,sc_c_id_index); Using where
1	SIMPLE	s	eq_ref	PRIMARY	PRIMARY	4	YSB.sc.s_id	1	

这里用到了intersect并集操作，即两个索引同时检索的结果再求并集，再看字段score和c_id的区分度，单从一个字段看，区分度都不是很大，从SC表检索，c_id=81检索的结果是70001,score=84的结果是39425。

而c_id=81 and score=84 的结果是897，即这两个字段联合起来的区分度是比较高的，因此建立联合索引查询效率将会更高，从另外一个角度看，该表的数据是300w，以后会更多，就索引存储而言，都是不小的数目，随着数据量的增加，索引就不能全部加载到内存，而是要从磁盘去读取，这样索引的个数越多，读磁盘的开销就越大，因此根据具体业务情况建立多列的联合索引是必要的，那么我们来试试吧。

```
alter table SC drop index sc_c_id_index;
alter table SC drop index sc_score_index;
create index sc_c_id_score_index on SC(c_id,score);
```

执行上述查询语句

消耗时间为：0.007s

这个速度还是可以接受的

执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	sc	ref	sc_c_id_score_index	sc_c_id_score_index	10	const,const	884	Using where
1	SIMPLE	s	eq_ref	PRIMARY	PRIMARY	4	YSB.sc.s_id	1	

该语句的优化暂时告一段落

总结：

- mysql嵌套子查询效率确实比较低
- 可以将其优化成连接查询
- 连接表时，可以先用where条件对表进行过滤，然后做表连接
(虽然mysql会对连表语句做优化)
- 建立合适的索引，必要时建立多列联合索引
- 学会分析sql执行计划，mysql会对sql进行优化，所以分析执行计划很重要

索引优化

上面讲到子查询的优化，以及如何建立索引，而且在多个字段索引时，分别对字段建立了单个索引

后面发现其实建立联合索引效率会更高，尤其是在数据量较大，单个列区分度不高的情况下。

单列索引

查询语句如下：

```
select * from user_test_copy where sex = 2 and type = 2 and age = 1
```

索引：

```
CREATE index user_test_index_sex on user_test_copy(sex);
CREATE index user_test_index_type on user_test_copy(type);
CREATE index user_test_index_age on user_test_copy(age);
```

分别对sex,type,age字段做了索引，数据量为300w

查询时间：0.415s

执行计划：

信息									
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user_test_copy	index_merge	user_test_index_sex,user_test_index_type,user_test_index_type, 5,5	(Null)	126			Using intersect(user_test_index_type,user_test_index_age); Using wh

发现type=index_merge

这是mysql对多个单列索引的优化，对结果集采用intersect并集操作

多列索引

我们可以在这3个列上建立多列索引，将表copy一份以便做测试：

```
create index user_test_index_sex_type_age on user_test(sex,type,age);
```

查询语句：

```
select * from user_test where sex = 2 and type = 2 and age = 10
```

执行时间：0.032s

快了10多倍，且多列索引的区分度越高，提高的速度也越多

执行计划：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user_test	ref	user_test_index_se	user_test_index_15	const,const,const	1530	Using where	

最左前缀

多列索引还有最左前缀的特性：

都会使用到索引，即索引的第一个字段sex要出现在where条件中

执行一下语句：

```
select * from user_test where sex = 2  
select * from user_test where sex = 2 and type = 2  
select * from user_test where sex = 2 and age = 10
```

索引覆盖

就是查询的列都建立了索引，这样在获取结果集的时候不用再去磁盘获取其它列的数据，直接返回索引数据即可

如：

```
select sex,type,age from user_test where sex = 2 and type = 2 and age = 10
```

执行时间：0.003s

要比取所有字段快的多

排序

```
select * from user_test where sex = 2 and type = 2 ORDER BY user_name
```

时间：0.139s

在排序字段上建立索引会提高排序的效率

```
create index user_name_index on user_test(user_name)
```

最后附上一些sql调优的总结，以后有时间再深入研究

- 列类型尽量定义成数值类型，且长度尽可能短，如主键和外键，类型字段等等
- 建立单列索引
- 根据需要建立多列联合索引

当单个列过滤之后还有很多数据，那么索引的效率将会比较低，即列的区分度较低，那么如果在多个列上建立索引，那么多个列的区分度就大多了，将会有显著的效率提高。

- 根据业务场景建立覆盖索引

- 多表连接的字段上需要建立索引

这样可以极大的提高表连接的效率

- where条件字段上需要建立索引
- 排序字段上需要建立索引
- 分组字段上需要建立索引
- Where条件上不要使用运算函数，以免索引失效

参考文章

<http://www.cnblogs.com/linfangshuhollowred/p/4430293.html>
<http://tech.meituan.com/mysql-index.html>
<http://www.cnblogs.com/Tool0/p/3634563.html>
<http://www.cnblogs.com/mliang/p/3637937.html>
<http://www.cnblogs.com/xwdreamer/archive/2012/07/19/2599494.html>
<http://www.cnblogs.com/ggjucheng/archive/2012/11/11/2765237.html>

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

[1. Spring Boot:启动原理解析](#)

[2. 一键下载 Pornhub 视频！](#)

[3. Spring Boot 多模块项目实践\(附打包方法\)](#)

[4. 一个女生不主动联系你还有机会吗?](#)

[5. 团队开发中 Git 最佳实践](#)



喜欢文章, 点个在看 

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

一步一步带你入门 MySQL 中的索引和锁

Java后端 2019-11-14

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | FrancisQ

链接 | juejin.im/post/5db19103e51d452a300b14c9

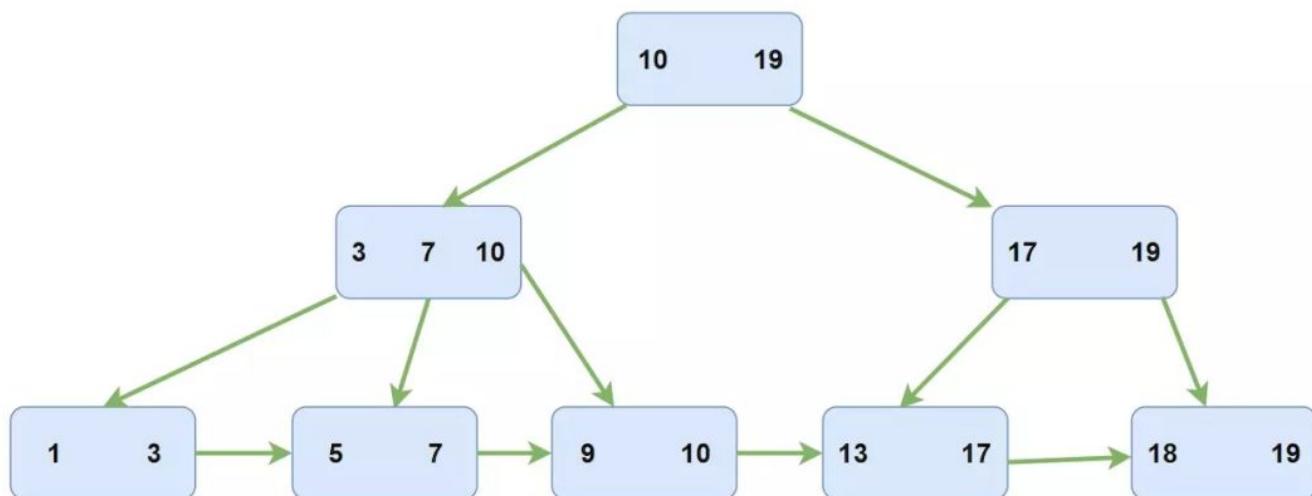
索引

索引常见的几种类型

索引常见的类型有哈希索引，有序数组索引，二叉树索引，跳表等等。本文主要探讨 MySQL 的默认存储引擎 InnoDB 的索引结构。

InnoDB的索引结构

在InnoDB中是通过一种多路搜索树——B+树实现索引结构的。在B+树中是只有叶子结点会存储数据，而且所有叶子结点会形成一个链表。而在InnoDB中维护的是一个双向链表。



你可能会有一个疑问，为什么使用 B+树 而不使用二叉树或者B树？

首先，我们知道访问磁盘需要访问到指定块中，而访问指定块是需要 盘片旋转 和 磁臂移动 的，这是一个比较耗时的过程，如果增加树高那么就意味着你需要进行更多次的磁盘访问，所以会采用n叉树。

而使用B+树是因为如果使用B树在进行一个范围查找的时候每次都会进行重新检索，而在B+树中可以充分利用叶子结点的链表。

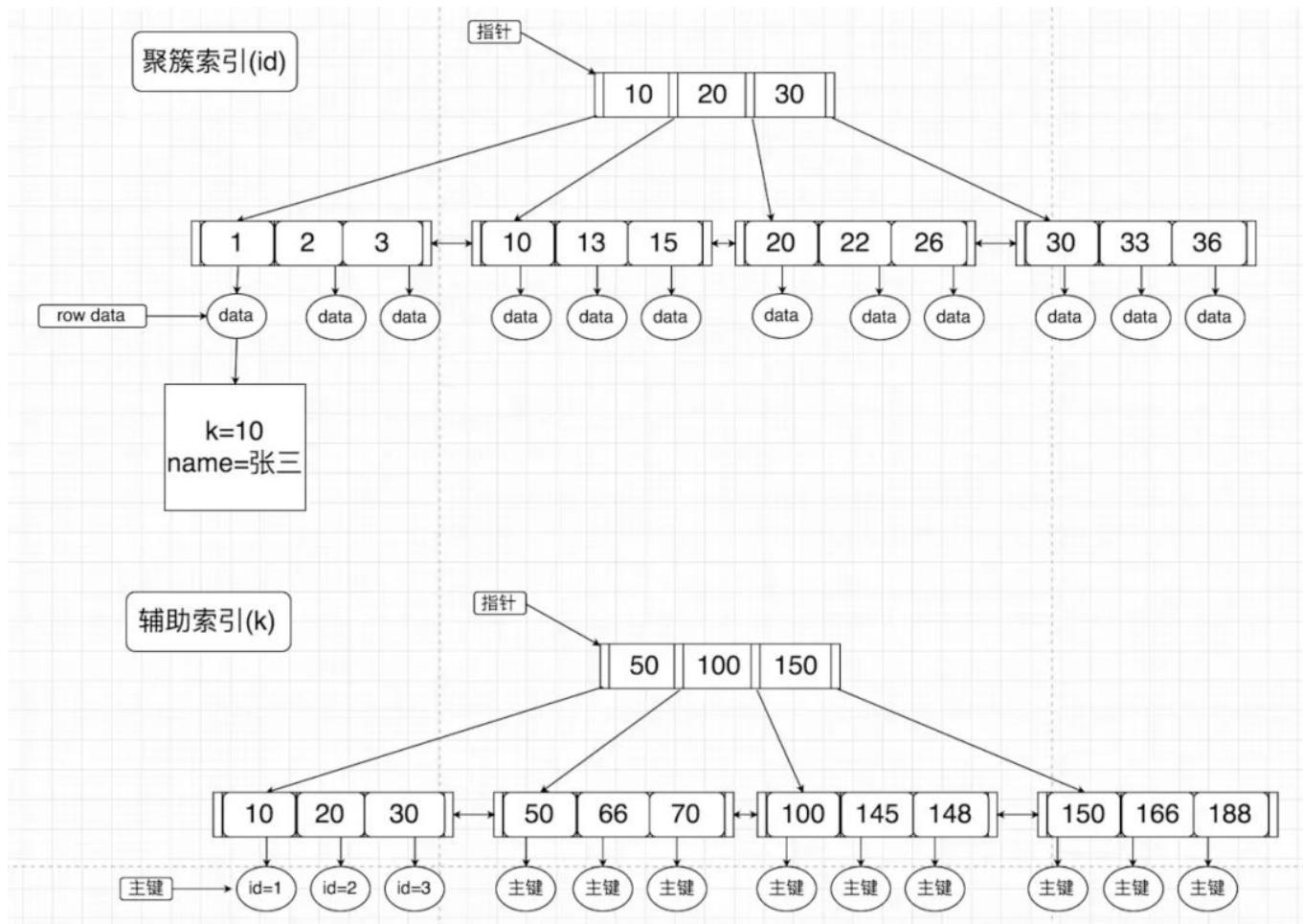
在建表的时候你可能会添加多个索引，而 InnoDB 会为每个索引建立一个 B+树 进行存储索引。

比如这个时候我们建立了一个简单的测试表

```
create table test(
    id int primary key,
    a int not null,
    name varchar,
    index(a)
)engine = InnoDB;
```

这个时候 InnoDB 就会为我们建立两个 B+索引树

一个是 主键 的 聚簇索引，另一个是 普通索引 的 辅助索引，这里我直接贴上 MySQL浅谈（索引、锁）这篇文章上面的贴图(因为我懒不想画图了。。。)



可以看到在辅助索引上面的叶子节点的值只是存了主键的值，而在主键的聚簇索引上的叶子节点才是存上了整条记录的值。

回表

所以这里就会引申出一个概念叫回表，比如这个时候我们进行一个查询操作

```
select name from test where a = 30;
```

我们知道因为条件 MySQL 是会走 a 的索引的，但是 a 索引上并没有存储 name 的值，此时我们就需要拿到相应 a 上的主键值，然后通过这个主键值去走 聚簇索引 最终拿到其中的name值，这个过程就叫回表。

我们来总结一下回表是什么？MySQL在辅助索引上找到对应的主键值并通过主键值在聚簇索引上查找所要的数据就叫回表。

索引维护

我们知道索引是需要占用空间的，索引虽能提升我们的查询速度但是也是不能滥用。

比如我们在用户表里用身份证号做主键，那么每个二级索引的叶子节点占用约20个字节，而如果用整型做主键，则只要4个字节，如果是长整型（bigint）则是8个字节。也就是说如果我用整型后面维护了4个g的索引列表，那么用身份证将会是20个g。

所以我们可以通过缩减索引的大小来减少索引所占空间。

当然B+树为了维护索引的有序性会在删除，插入的时候进行一些必要的维护（在InnoDB中删除会将节点标记为“可复用”以减少对结构的变动）。

比如在增加一个节点的时候可能会遇到数据页满了的情况，这个时候就需要做页的分裂，这是一个比较耗时的工作，而且页的分裂还会导致数据页的利用率变低，比如原来存放三个数据的数据页再次添加一个数据的时候需要做页分裂，这个时候就会将现有的四个数据分配到两个数据页中，这样就减少了数据页利用率。

覆盖索引

上面提到了回表，而有时候我们查辅助索引的时候就已经满足了我们需要查的数据，这个时候InnoDB就会进行一个叫覆盖索引的操作来提升效率，减少回表。

比如这个时候我们进行一个select操作

```
select id from test where a = 1;
```

这个时候很明显我们走了a的索引直接能获取到id的值，这个时候就不需要进行回表，我们这个时候就使用了覆盖索引。

简单来说覆盖索引就是当我们走辅助索引的时候能获取到我们所需要的数据的时候不需要再次进行回表操作的操作。

联合索引

这个时候我们新建一个学生表

```
CREATE TABLE `stu` (
  `id` int(11) NOT NULL,
  `class` int(11) DEFAULT NULL,
  `name` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `class_name` (`class`, `name`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

我们使用class(班级号)和name做一个联合索引，你可能会问这个联合索引有什么用呢？我们可以结合着上面的覆盖索引去理解，比如这个时候我们有一个需求，我们需要通过班级号去找对应的学生姓名。

```
select name from stu where class = 102;
```

这个时候我们就可以直接在辅助索引上查找到学生姓名而不需要再次回表。

总的来说，设计好索引，充分利用覆盖索引能很大提升检索速度。

最左前缀原则

这个是以联合索引作为基础的，是一种联合索引的匹配规则。

这个时候，我们将上面的需求稍微变动一下，这时我们有个学生迟到，但是他在门卫记录信息的时候只写了自己的名字张三而没

有写班级，所以我们需要通过学生姓名去查找相应的班级号。

```
select class from stu where name = '张三';
```

这个时候我们就不会走我们的联合索引了，而是进行了全表扫描。

为什么？因为 最左匹配原则。我们可以画一张简单的图来理解一下。

我们可以看到整个索引设计就是这么设计的，所以我们需要查找的时候也需要遵循着这个规则，如果我们直接使用name，那么 InnoDB 是不知道我们需要干什么的。

当然最左匹配原则还有这些规则

- 全值匹配的时候优化器会改变顺序，也就是说你全值匹配时的顺序和原先的联合索引顺序不一致没有关系，优化器会帮你调好。
- 索引匹配从最左边的地方开始，如果没有则会进行全表扫描，比如你设计了一个(a,b,c)的联合索引，然后你可以使用(a), (a,b),(a,b,c) 而你使用 (b),(b,c),(c) 就用不到索引了。
- 遇到范围匹配会取消索引。比如这个时候你进行一个这样的 select 操作

```
select * from stu where class > 100 and name = '张三';
```

这个时候 InnoDB 就会放弃索引而进行全表扫描，因为这个时候 InnoDB 会不知道怎么进行遍历索引，所以进行全表扫描。

索引下推

我给你挖了个坑。刚刚的操作在 MySQL5.6 版本以前是需要进行回表的，但是5.6之后的版本做了一个叫 索引下推 的优化。

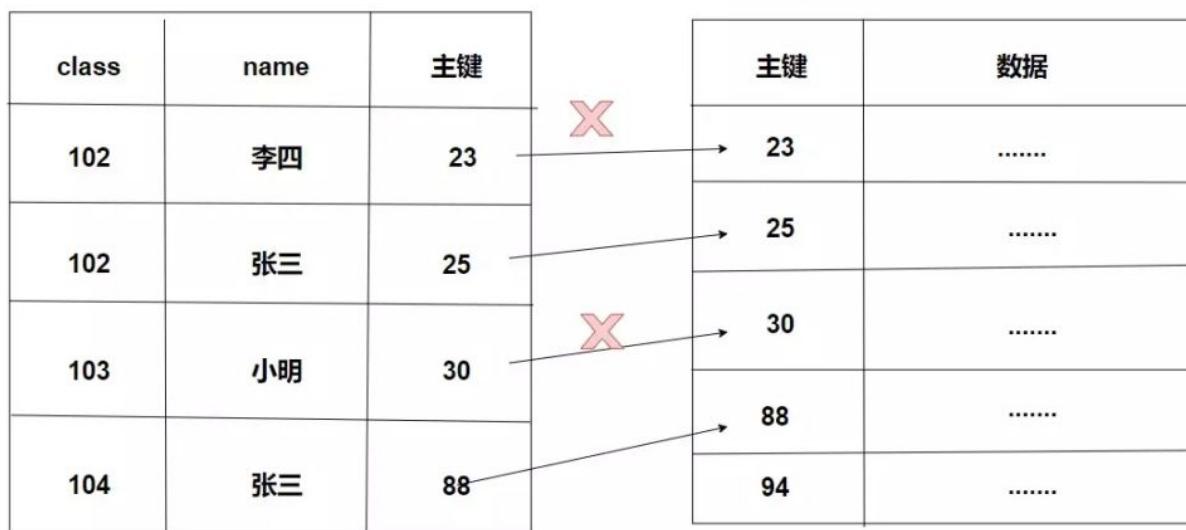
```
select * from stu where class > 100 and name = '张三';
```

如何优化的呢？因为刚刚的最左匹配原则我们放弃了索引，后面我们紧接着会通过回表进行判断 name，这个时候我们所要做的操作应该是这样的

class	name	主键
102	李四	23
102	张三	25
103	小明	30
104	张三	88

但是有了索引下推之后就变成这样了，此时 "李四" 和 "小明" 这两个不会再进行回表。

主键索引



因为这里匹配了后面的name = 张三，也就是说，如果最左匹配原则因为范围查询终止了，InnoDB还是会索引下推来优化性能。

一些最佳实践

哪些情况需要创建索引？

- 频繁作为查询条件的字段应创建索引。
- 多表关联查询的时候，关联字段应该创建索引。
- 查询中的排序字段，应该创建索引。
- 统计或者分组字段需要创建索引。

哪些情况不需要创建索引

- 表记录少。
- 经常增删改查的表。
- 频繁更新的字段。
- where 条件使用不高的字段。
- 字段很大的时候。

其他

- 尽量选择区分度高的列作为索引。
- 不要对索引进行一些函数操作，还应注意隐式的类型转换和字符编码转换。
- 尽可能的扩展索引，不要新建立索引。比如表中已经有了a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。
- 多考虑覆盖索引，索引下推，最左匹配。

锁

全局锁

MySQL提供了一个加全局读锁的方法，命令是 `Flush tables with read lock (FTWRL)`。当你需要让整个库处于只读状态的时候，可以使用这个命令，之后其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。

一般会在进行全库逻辑备份的时候使用，这样就能确保其他线程不能对该数据库做更新操作。

在 MVCC 中提供了获取一致性视图的操作使得备份变得非常简单，如果想了解 MVCC 可以参考

<https://juejin.im/post/5da8493ae51d4524b25add55>

表锁

MDL(Meta Data Lock)元数据锁

MDL锁用来保证只有一个线程能对该表进行表结构更改。

怎么说呢？MDL分为MDL写锁和MDL读锁，加锁规则是这样的

- 当线程对一个表进行CRUD操作的时候会加MDL读锁
- 当线程对一个表进行表结构更改操作的时候会加MDL写锁
- 写锁和读锁，写锁和写锁互斥，读锁之间不互斥

lock tables xxx read/write;

这是给一个表设置读锁和写锁的命令，如果在某个线程A中执行`lock tables t1 read, t2 write;`这个语句，则其他线程写t1、读写t2的语句都会被阻塞。同时，线程A在执行`unlock tables`之前，也只能执行读t1、读写t2的操作。连写t1都不允许，自然也不能访问其他表。

这种表锁是一种处理并发的方式，但是在InnoDB中常用的是行锁。

行锁

我们知道在5.5版本以前 MySQL 的默认存储引擎是 MyISAM，而 MyISAM 和 InnoDB 最大的区别就是两个

- 事务
- 行锁

其中行锁是我们今天的主题，如果不了解事务可以去补习一下。

其实行锁就是两个锁，你可以理解为写锁(排他锁 X锁)和读锁(共享锁 S锁)

- 共享锁(S锁)：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。也叫做读锁：读锁是共享的，多个客户可以同时读取同一个资源，但不允许其他客户修改。
- 排他锁(X锁)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。也叫做写锁：写锁是排他的，写锁会阻塞其他的写锁和读锁。

而行锁还会引起一个一个很头疼的问题，那就是死锁。

如果事务A对行100加了写锁，事务B对行101加了写锁，此时事务A想要修改行101而事务B又想修改行100，这样占有且等待就导致了死锁问题，而面对死锁问题就只有检测和预防了。

next-key锁

MVCC 和行锁是无法解决 幻读 问题的，这个时候 InnoDB 使用了一个叫 GAP锁(间隙锁) 的东西，它配合 行锁 形成了 next-key 锁，解决了幻读的问题。

但是因为它的加锁规则，又导致了扩大了一些加锁范围从而减少数据库并发能力。具体的加锁规则如下：

- 加锁的基本单位是next-key lock 就是行锁和GAP锁结合。
- 查找过程中访问到的对象就会加锁。
- 索引上的等值查询，给唯一索引加锁的时候，next-key lock退化为行锁。
- 索引上的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next-key lock退化为间隙锁。
- 唯一索引上的范围查询会访问到不满足条件的第一个值为止。

MVCC 解决幻读的思路比较复杂，这里就不做过多的验证。

总结

对于 MySQL 的索引来说，我给了很多最佳实践，其实这些最佳实践都是从原理来的，而 InnoDB 其实就是一个改进版的 B+树，还有存储索引的结构。弄懂了这些你就会得心应手起来。

而对于 MySQL 的锁，主要就是在行锁方面，InnoDB 其实就是使用了 行锁，MVCC还有next-key锁来实现事务并发控制的。

而对于MySQL中最重要的其实就是 锁和索引 了，因为内容太多这篇文章仅仅做一些介绍和简单的分析，如果想深入了解可以查看相应的文章。

- E N D -

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



扫一扫上面的二维码图案，加我微信

推荐阅读

1. [SSM 实现支付宝扫码支付功能 \(图文详解\)](#)
2. [Spring Boot 微信点餐系统](#)
3. [Spring MVC 到 Spring Boot 的简化之路](#)
4. [12306 的架构到底有多牛逼？](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章，点个在看 

写一手好 SQL 很有必要

编码砖家 Java后端 1月2日

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

来源 | 编码砖家

链接 | cnblogs.com/xiaoyangjia/p/11267191.html

前言

博主负责的项目主要采用阿里云数据库MySQL，最近频繁出现慢SQL告警，**执行时间最长的竟然高达5分钟**。导出日志后分析，主要原因竟然是**没有命中索引和没有分页处理**。

其实这是非常低级的错误，我不禁后背一凉，团队成员的技术水平亟待提高啊。改造这些SQL的过程中，总结了一些经验分享给大家，如果有错误欢迎批评指正。

MySQL性能

最大数据量

抛开数据量和并发数，谈性能都是耍流氓。MySQL没有限制单表最大记录数，它取决于操作系统对文件大小的限制。

文件系统	单文件大小限制
FAT32	最大4G
NTFS	最大64GB
NTFS5.0	最大2TB
EXT2	块大小为1024字节，文件最大容量16GB；块大小为4096字节，文件最大容量2TB
EXT3	块大小为4KB，文件最大容量为4TB
EXT4	理论可以大于16TB

《阿里巴巴Java开发手册》提出单表行数超过500万行或者单表容量超过2GB，才推荐分库分表。性能由综合因素决定，抛开业务复杂度，影响程度依次是硬件配置、MySQL配置、数据表设计、索引优化。500万这个值仅供参考，并非铁律。博主曾经操作过超过4亿行数据的单表，分页查询最新的20条记录耗时0.6秒，SQL语句大致是 `select field_1,field_2 from table where id < #{prePageMinId} order by id desc limit 20`，`prePageMinId`是上一页数据记录的最小ID。虽然当时查询速度还凑合，随着数据不断增长，有朝一日必定不堪重负。分库分表是个周期长而风险高的大活儿，应该尽可能在当前结构上优化，比如升级硬件、迁移历史数据等等，实在没辙了再分。对分库分表感兴趣的同学可以阅读分库分表的基本思想。

最大并发数

并发数是指同一时刻数据库能处理多少个请求，由`max_connections`和`max_user_connections`决定。

****max_connections**是指MySQL实例的最大连接数，上限值是16384，**max_user_connections**是指每个数据库用户的最大连接数。MySQL会为每个连接提供缓冲区，意味着消耗更多的内存。如果连接数设置太高硬件吃不消，太低又不能充分利用硬件。一般要求两者比值超过10%，计算方法如下：

$$\text{max_used_connections} / \text{max_connections} * 100\% = 3/100 * 100\% \approx 3\%$$

查看最大连接数与响应最大连接数：

```
show variables like '%max_connections%'; show variables like '%max_user_connections%';
```

在配置文件my.cnf中修改最大连接数

```
[mysqld]max_connections = 100max_used_connections = 20
```

查询耗时0.5秒

建议将单次查询耗时控制在0.5秒以内，0.5秒是个经验值，源于用户体验的**3秒原则**。如果用户的操作3秒内没有响应，将会厌烦甚至退出。响应时间=客户端UI渲染耗时+网络请求耗时+应用程序处理耗时+查询数据库耗时，0.5秒就是留给数据库1/6的处理时间。

实施原则

相比NoSQL数据库，MySQL是个娇气脆弱的家伙。它就像体育课上的女同学，一点纠纷就和同学闹别扭(扩容难)，跑两步就气喘吁吁(容量小并发低)，常常身体不适要请假(SQL约束太多)。如今大家都会搞点分布式，应用程序扩容比数据库要容易得多，所以实施原则是**数据库少干活，应用程序多干活**。

- 充分利用但不滥用索引，须知索引也消耗磁盘和CPU。
- 不推荐使用数据库函数格式化数据，交给应用程序处理。
- 不推荐使用外键约束，用应用程序保证数据准确性。
- 写多读少的场景，不推荐使用唯一索引，用应用程序保证唯一性。
- 适当冗余字段，尝试创建中间表，用应用程序计算中间结果，用空间换时间。
- 不允许执行极度耗时的事务，配合应用程序拆分成更小的事务。
- 预估重要数据表（比如订单表）的负载和数据增长态势，提前优化。

数据表设计

数据类型

数据类型的选择原则：更简单或者占用空间更小。

- 如果长度能够满足，整型尽量使用tinyint、smallint、medium_int而非int。
- 如果字符串长度确定，采用char类型。
- 如果varchar能够满足，不采用text类型。
- 精度要求较高的使用decimal类型，也可以使用BIGINT，比如精确两位小数就乘以100后保存。
- 尽量采用timestamp而非datetime。

类型	占据字节	描述
datetime	8字节	'1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'
timestamp	4字节	'1970-01-01 00:00:01.000000' to '2038-01-19 03:14:07.999999'

相比datetime, timestamp占用更少的空间，以UTC的格式储存自动转换时区。

避免空值

MySQL中字段为NULL时依然占用空间，会使索引、索引统计更加复杂。从NULL值更新到非NULL无法做到原地更新，容易发生索引分裂影响性能。尽可能将NULL值用有意义的值代替，也能避免SQL语句里面包含 `is not null` 的判断。

text类型优化

由于text字段储存大量数据，表容量会很早涨上去，影响其他字段的查询性能。建议抽取出来放在子表里，用业务主键关联。

索引优化

索引分类

1. 普通索引：最基本的索引。
2. 组合索引：多个字段上建立的索引，能够加速复合查询条件的检索。
3. 唯一索引：与普通索引类似，但索引列的值必须唯一，允许有空值。
4. 组合唯一索引：列值的组合必须唯一。
5. 主键索引：特殊的唯一索引，用于唯一标识数据表中的某一条记录，不允许有空值，一般用primary key约束。
6. 全文索引：用于海量文本的查询，MySQL5.6之后的InnoDB和MyISAM均支持全文索引。由于查询精度以及扩展性不佳，更多的企业选择Elasticsearch。

索引优化

1. 分页查询很重要，如果查询数据量超过30%，MySQL不会使用索引。
2. 单表索引数不超过5个、单个索引字段数不超过5个。
3. 字符串可使用前缀索引，前缀长度控制在5-8个字符。
4. 字段唯一性太低，增加索引没有意义，如：是否删除、性别。
5. 合理使用覆盖索引，如下所示：

```
select loginname, nickname from member where login_name = ?
```

loginname, nickname两个字段建立组合索引，比login_name简单索引要更快

SQL优化

分批处理

博主小时候看到鱼塘挖开小口子放水，水面有各种漂浮物。浮萍和树叶总能顺利通过出水口，而树枝会挡住其他物体通过，有时还会卡住，需要人工清理。MySQL就是鱼塘，最大并发数和网络带宽就是出水口，用户SQL就是漂浮物。不带分页参数的查询或者影响大量数据的update和delete操作，都是树枝，我们要把它打散分批处理，举例说明：业务描

述：更新用户所有已过期的优惠券为不可用状态。SQL语句：`update status=0 FROM coupon WHERE expire_date <= #{currentDate} and status=1;` 如果大量优惠券需要更新为不可用状态，执行这条SQL可能会堵死其他SQL，分批处理伪代码如下：

```
int pageNo = 1;  
int PAGE_SIZE = 100;  
  
while(true){  
    List<Integer> batchIdList = queryList('select id FROM `coupon` WHERE expire_date <= #{currentDate} and status = 1 limit #[(pageNo-1) * PAGE_SIZE, PAGE_SIZE]');  
    if(CollectionUtils.isEmpty(batchIdList)){  
        return;  
    }  
    update('update status = 0 FROM `coupon` where status = 1 and id in #{batchIdList}')  
    pageNo++;  
}
```

操作符<>优化

通常<>操作符无法使用索引，举例如下，查询金额不为100元的订单：`select id from orders where amount != 100;` 如果金额为100的订单极少，这种数据分布严重不均的情况下，有可能使用索引。鉴于这种不确定性，采用union聚合搜索结果，改写方法如下：

```
(select id from orders where amount > 100) union all(select id from orders where amount < 100 and amount > 0)
```

OR优化

在Innodb引擎下or无法使用组合索引，比如：

```
select id, product_name from orders where mobile_no = '13421800407' or user_id = 100;
```

OR无法命中mobile_no + userid的组合索引，可采用union，如下所示：

```
(select id, product_name from orders where mobile_no = '13421800407') union(select id, product_name from orders where user_id = 100);
```

此时id和product_name字段都有索引，查询才最高效。

IN优化

1. IN适合主表大子表小，EXIST适合主表小子表大。由于查询优化器的不断升级，很多场景这两者性能差不多一样了。

2. 尝试改为join查询，举例如下：

```
select id from orders where user_id in (select id from user where level = 'VIP');
```

采用JOIN如下所示：

```
select o.id from orders o left join user u on o.user_id = u.id where u.level = 'VIP';
```

不做列运算

通常在查询条件列运算会导致索引失效，如下所示：查询当日订单

```
select id from order where date_format(create_time, '%Y-%m-%d') = '2019-07-01';
```

date_format函数会导致这个查询无法使用索引，改写后：

```
select id from order where create_time between '2019-07-01 00:00:00' and '2019-07-01 23:59:59';
```

避免Select all

如果不查询表中所有的列，避免使用 `SELECT *`，它会进行全表扫描，不能有效利用索引。

Like优化

like用于模糊查询，举个例子（field已建立索引）：

```
SELECT column FROM table WHERE field like '%keyword%';
```

这个查询未命中索引，换成下面的写法：

```
SELECT column FROM table WHERE field like 'keyword%';
```

去除了前面的%查询将会命中索引，但是产品经理一定要前后模糊匹配呢？全文索引fulltext可以尝试一下，但 Elasticsearch才是终极武器。

欢迎关注公众号：Java后端

Join优化

join的实现是采用Nested Loop Join算法，就是通过驱动表的结果集作为基础数据，通过该结果集作为过滤条件到下一个表中循环查询数据，然后合并结果。如果有多个join，则将前面的结果集作为循环数据，再次到后一个表中查询数据。

1. 驱动表和被驱动表尽可能增加查询条件，满足ON的条件而少用Where，用小结果集驱动大结果集。
2. 被驱动表的join字段上加上索引，无法建立索引的时候，设置足够的Join Buffer Size。
3. 禁止join连接三个以上的表，尝试增加冗余字段。

Limit优化

limit用于分页查询时越往后翻性能越差，解决的原则：**缩小扫描范围**，如下所示：

```
select * from orders order by id desc limit 100000,10 耗时0.4秒select * from orders order by id desc limit 1000000,10耗时5.2秒
```

先筛选出ID缩小查询范围，写法如下：

```
select * from orders where id > (select id from orders order by id desc limit 1000000, 1) order by id desc limit 0,10耗时0.5秒
```

如果查询条件仅有主键ID，写法如下：

```
select id from orders where id between 1000000 and 1000010 order by id desc耗时0.3秒
```

如果以上方案依然很慢呢？只好用游标了，感兴趣的朋友阅读JDBC使用游标实现分页查询的方法

其他数据库

作为一名后端开发人员，务必精通作为存储核心的MySQL或SQL Server，也要积极关注NoSQL数据库，他们已经足够成熟并被广泛采用，能解决特定场景下的性能瓶颈。

分类	数据库	特性
键值型	Memcache	用于内容缓存，大量数据的高访问负载
键值型	[Redis](https://redis.io/)	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	HBase	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	MongoDb	知名文档型数据库，也可以用于缓存
文档型	CouchDB	Apache的开源项目，专注于易用性，支持REST API
文档型	SequoiaDB	国内知名文档型数据库
图形	Neo4J	用于社交网络构建关系图谱，推荐系统等

- END -

推荐阅读

1. 还没抢到票吗？
2. 关于 CPU 的一些基本知识总结
3. 发布没有答案的面试题，都是耍流氓
4. 什么是一致性 Hash 算法？
5. 团队开发中 Git 最佳实践



声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

几种常用的 MySQL 图形化管理工具

陈万洲的专栏 Java后端 2019-11-27

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 陈万洲的专栏

来源 | blog.csdn.net/veloi/article/details/81386904

MySQL的管理维护工具非常多，除了系统自带的命令行管理工具之外，还有许多其他的图形化管理工具，这里我介绍几个经常使用的MySQL图形化管理工具，供大家参考。

MySQL是一个非常流行的小型关系型数据库管理系统，2008年1月16号被Sun公司收购。目前MySQL被广泛地应用在Internet上的中小型 网站中。由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，许多中小型网站为了降低网站总体拥有成本而选择了MySQL作为网站数据库。

1、phpMyAdmin

The screenshot shows the phpMyAdmin interface for the 'wordpress' database. The left sidebar lists the database structure with tables like wp_comments, wp_links, wp_options, wp_postmeta, wp_posts, wp_terms, wp_term_relationships, wp_term_taxonomy, wp_usermeta, and wp_users. The main area displays a table of 10 tables with their respective record counts, types, and sizes. The table structure includes columns for '表' (Table), '操作' (Operations), '记录数' (Records), '类型' (Type), '整理' (Collation), '大小' (Size), and '多余' (Extra).

表	操作	记录数	类型	整理	大小	多余
wp_comments	导出	0	MyISAM	utf8_general_ci	1.0 KB	-
wp_links	导出	1	MyISAM	utf8_general_ci	4.1 KB	-
wp_options	导出	142	MyISAM	utf8_general_ci	430.5 KB	-
wp_postmeta	导出	4,586	MyISAM	utf8_general_ci	728.4 KB	-
wp_posts	导出	1,532	MyISAM	utf8_general_ci	1.2 MB	-
wp_terms	导出	7	MyISAM	utf8_general_ci	5.2 KB	-
wp_term_relationships	导出	1,540	MyISAM	utf8_general_ci	90.6 KB	-
wp_term_taxonomy	导出	7	MyISAM	utf8_general_ci	3.3 KB	-
wp_usermeta	导出	10	MyISAM	utf8_general_ci	7.4 KB	-
wp_users	导出	1	MyISAM	utf8_general_ci	4.1 KB	-
10 个表		总计	7,826	MyISAM	utf8_general_ci	2.4 MB 0 字节

phpMyAdmin是最常用的MySQL维护工具，是一个用PHP开发的基于Web方式架构在网站主机上的MySQL管理工具，支持中文，管理数据库非常方便。不足之处在于对大数据库的备份和恢复不方便。

2、MySQLDumper

The screenshot shows the MySQLDumper Home page. It displays version information (Version 1.22) and system details (OS: WINNT, MySQL Version: 5.0.45-community-nt-log). The 'Status Information' section shows the MySQLDumper version (1.22), OS (WINNT), MySQL version (5.0.45), PHP version (5.2.3), and various MySQL extensions. The 'MySQL Dumper Information' section shows the location as "localhost" (F:/wwwroot/php/mysqlDumper/), the actual database as "wordpress", and a warning about creating directory protection. The 'History' section shows 14 backups (255.46 MB) and the last backup from 03.03.2009 06:08, which is "cnbaodao_2009_03_02_22_01.sql.gz".

MySQLDumper使用PHP开发的MySQL数据库备份恢复程序，解决了使用PHP进行大数据库备份和恢复的问题，数百兆的数据库

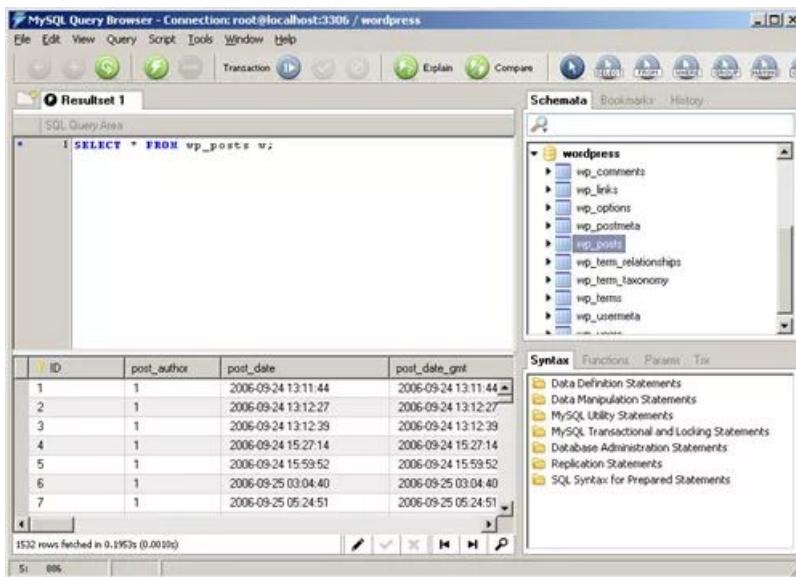
都可以方便的备份恢复，不用担心网速太慢导致中间中断的问题，非常方便易用。这个软件是德国人开发的，还没有中文语言包。

3、Navicat



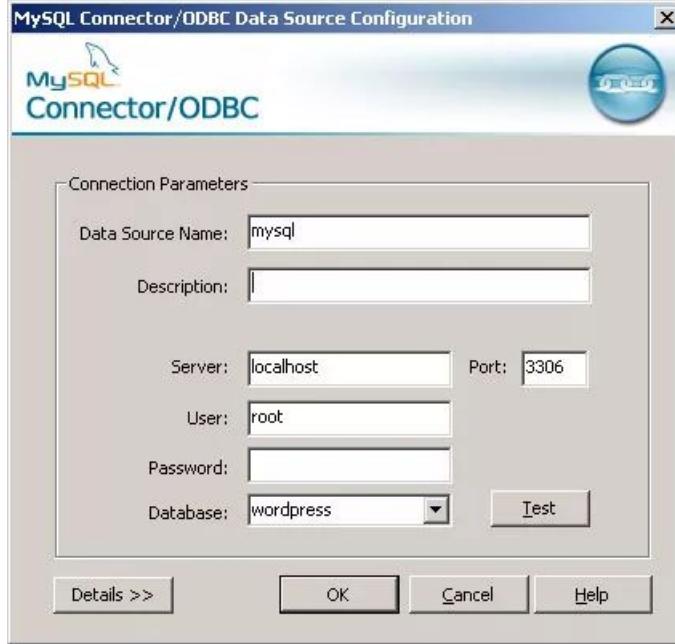
Navicat是一个桌面版MySQL数据库管理和开发工具。和微软SQLServer的管理器很像，易学易用。Navicat使用图形化的用户界面，可以让用户使用和管理更为轻松。支持中文，有免费版本提供。

4、MySQL GUI Tools



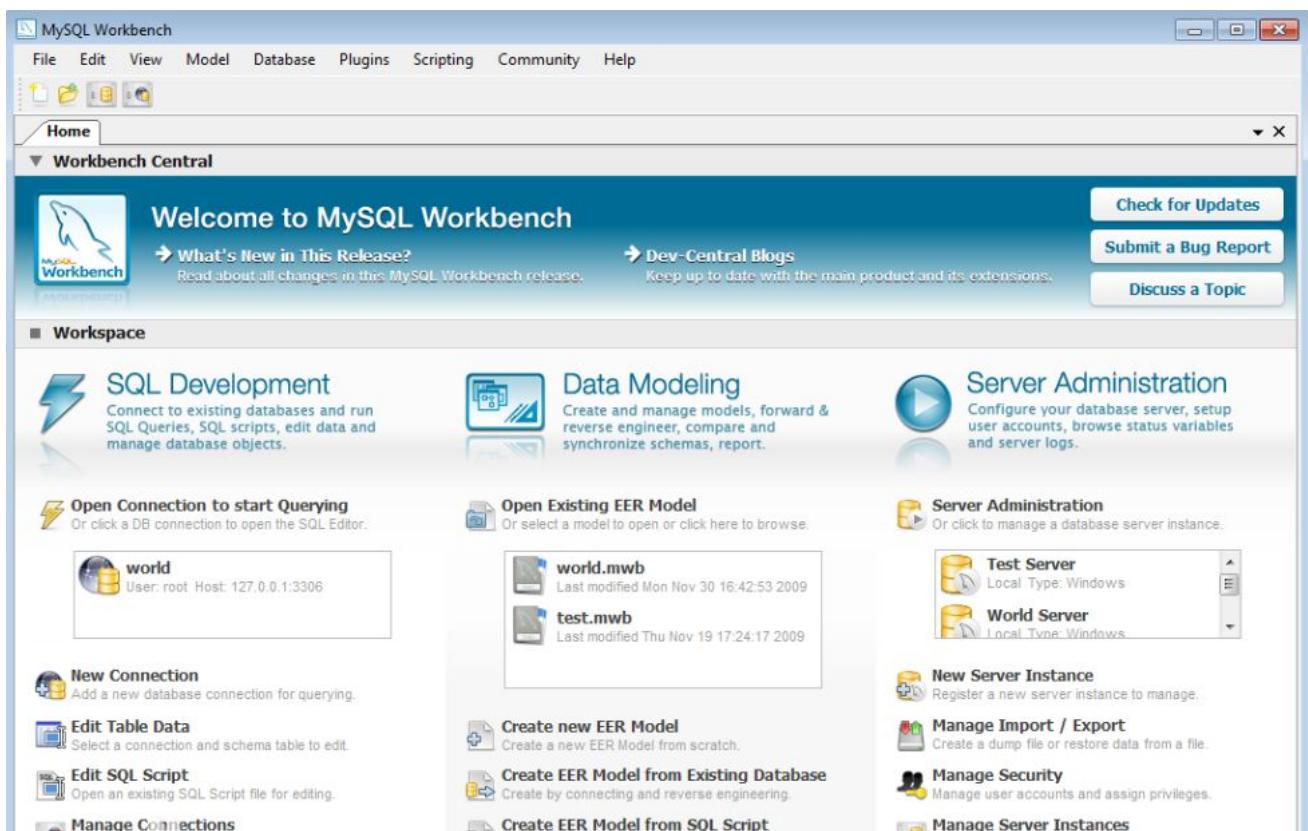
MySQL GUI Tools是MySQL官方提供的图形化管理工具，功能很强大，值得推荐，可惜的是没有中文界面。

5、MySQL ODBC Connector



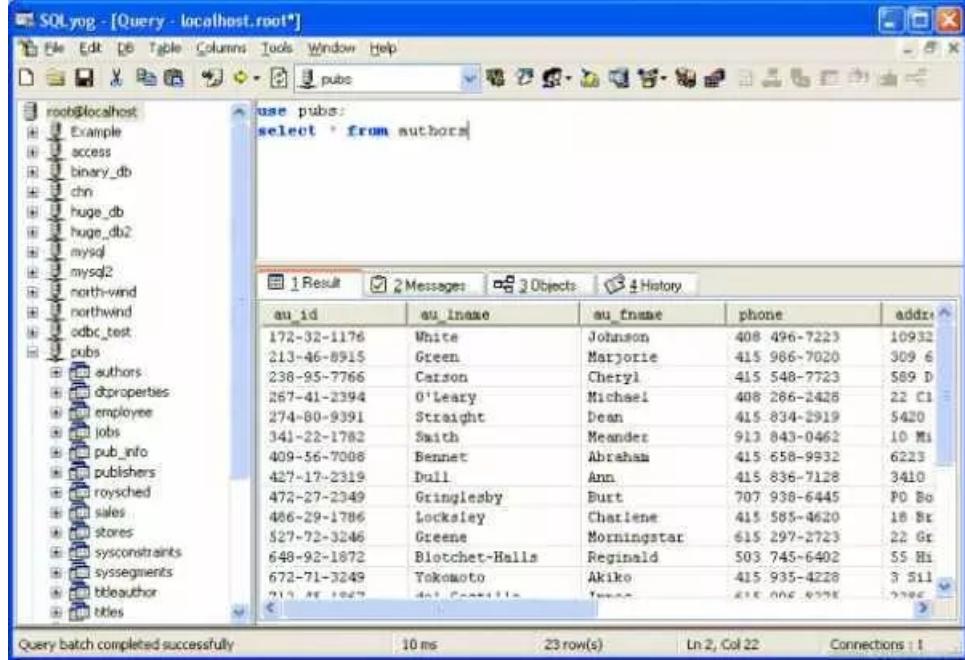
MySQL官方提供的ODBC接口程序，系统安装了这个程序之后，就可以通过ODBC来访问MySQL，这样就可以实现SQLServer、Access和MySQL之间的数据转换，还可以支持ASP访问MySQL数据库。

6、MySQL Workbench



MySQL Workbench是一个统一的可视化开发和管理平台，该平台提供了许多高级工具，可支持数据库建模和设计、查询开发和测试、服务器配置和监视、用户和安全管理、备份和恢复自动化、审计数据检查以及向导驱动的数据库迁移。MySQL Workbench是MySQL AB发布的可视化的数据库设计软件，它的前身是FabForce公司的DDesigner 4。MySQL Workbench为数据库管理员、程序开发者和系统规划师提供可视化设计、模型建立、以及数据库管理功能。它包含了用于创建复杂的数据建模EER模型，正向和逆向数据库工程，也可以用于执行通常需要花费大量时间和需要的难以变更和管理的文档任务。MySQL工作台可在Windows, Linux和Mac上使用。

7、SQLyog



SQLyog 是一个易于使用的、快速而简洁的图形化管理MYSQL数据库的工具，它能够在任何地点有效地管理你的数据库。

SQLyog是业界著名的Webyog公司出品的一款简洁高效、功能强大的图形化MySQL数据库管理工具。使用SQLyog可以快速直观地让您从世界的任何角落通过网络来维护远端的MySQL数据库。

SQLyog下载地址：https://pan.baidu.com/s/1_O0R-fUvSHFtgc84xZbJxg

SQLyog破解方法：<https://blog.csdn.net/veloi/article/details/80716375>

8、MySQL-Front

id	agent_id	type	affect_money	account_money	freeze_money	attach	info
6	14	1	14	14	0	13800	恒宇网
7	14	1	7	21	0	6500	老街网
8	14	1	24	45	0	24100	恒宇网
9	14	1	11	56	0	11300	老街网
10	14	1	16	72	0	15800	恒宇网
11	14	1	8	80	0	8200	老街网
12	14	1	13	93	0	13300	恒宇网
13	14	1	6	99	0	5600	老街网
14	14	1	19	118	0	18900	恒宇网
15	14	1	15	133	0	14700	老街网
16	14	1	19	152	0	19400	恒宇网
17	14	1	9	161	0	8800	老街网
18	14	1	20	181	0	19600	恒宇网
19	14	1	5	186	0	5200	老街网
20	11	1	1	1	0	900	泡泡网
21	14	1	17	210	0	17300	恒宇网
22	14	1	16	210	0	16200	泡泡网

小巧的管理Mysql的应用程序.主要特性包括多文档界面,语法突出,拖拽方式的数据库和表格,可编辑/可增加/删除的域,可编辑/可插入/删除的记录,可显示的成员,可执行的SQL 脚本,提供与外程序接口,保存数据到CSV文件等.

下载地址：<http://www.mysqlfront.de/>

9、dbForge Studio for MySQL

dbForge Studio for MySQL 是一款强大的工具,专门用来自动化和简单化MySQL的工作.它提供了一种简单的方式来探讨和维护现有的数据库,设计复合的SQL语句,以不同的方式查询和操作数据.

下载地址:

<https://www.devart.com/dbforge/mysql/studio/download.html#anchorDowload>

【END】

如果看到这里,说明你喜欢这篇文章,请[转发、点赞](#)。微信搜索「web_resource」,关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. [一文掌握 Redis 常用知识点 - 图文结合](#)
2. [面试官:讲一下 Mybatis 初始化原理](#)
3. [我们再来聊一聊 Java 的单例吧](#)
4. [我采访了一位 Pornhub 工程师](#)
5. [团队开发中 Git 最佳实践](#)



喜欢文章，点个在看 

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

如何去写一手好SQL？

编码砖家 Java后端 2019-10-02

点击上方 Java后端, 选择“**设为星标**”

优质文章, 及时送达

作者 | 编码砖家

链接 | cnblogs.com/xiaoyangjia/p/11267191.html

上篇 | [女友半夜不回家, 窃听了她手机后得知我被绿了](#)

目录

■ MySQL性能

- 最大数据量
- 最大并发数
- 查询耗时0.5秒
- 实施原则

■ 数据表设计

- 数据类型
- 避免空值
- text类型

■ 索引优化

- 索引分类
- 优化原则

■ SQL优化

- 分批处理
- 不做列运算
- 避免Select *
- 操作符<>优化
- OR优化
- IN优化
- LIKE优化
- JOIN优化
- LIMIT优化

■ 其他数据库

博主负责的项目主要采用阿里云数据库MySQL，最近频繁出现慢SQL告警，执行时间最长的竟然高达5分钟。导出日志后分析，主要原因竟然是没有命中索引和没有分页处理。其实这是非常低级的错误，我不禁后背一凉，团队成员的技术水平亟待提高啊。改造这些SQL的过程中，总结了一些经验分享给大家，如果有错误欢迎批评指正。

[MySQL性能](#)

最大数据量

抛开数据量和并发数，谈性能都是耍流氓。MySQL没有限制单表最大记录数，它取决于操作系统对文件大小的限制。

文件系统	单文件大小限制
FAT32	最大4G
NTFS	最大64GB
NTFS5.0	最大2TB
EXT2	块大小为1024字节，文件最大容量16GB；块大小为4096字节，文件最大容量2TB
EXT3	块大小为4KB，文件最大容量为4TB
EXT4	理论可以大于16TB

《阿里巴巴Java开发手册》提出单表行数超过500万行或者单表容量超过2GB，才推荐分库分表。性能由综合因素决定，抛开业务复杂度，影响程度依次是硬件配置、MySQL配置、数据表设计、索引优化。500万这个值仅供参考，并非铁律。微信搜索 web_resource 关注获取更多推送。

博主曾经操作过超过4亿行数据的单表，分页查询最新的20条记录耗时0.6秒，SQL语句大致是 `select field_1,field_2 from table where id < #{prePageMinId} order by id desc limit 20`，`prePageMinId` 是上一页数据记录的最小ID。

虽然当时查询速度还凑合，随着数据不断增长，有朝一日必定不堪重负。分库分表是个周期长而风险高的大活儿，应该尽可能在当前结构上优化，比如升级硬件、迁移历史数据等等，实在没辙了再分。对分库分表感兴趣的同学可以阅读分库分表的基本思想。

最大并发数

并发数是指同一时刻数据库能处理多少个请求，由 `max_connections` 和 `max_user_connections` 决定。`max_connections` 是指 MySQL 实例的最大连接数，上限值是 16384，`max_user_connections` 是指每个数据库用户的最大连接数。

MySQL 会为每个连接提供缓冲区，意味着消耗更多的内存。如果连接数设置太高硬件吃不消，太低又不能充分利用硬件。一般要求两者比值超过 10%，计算方法如下：

```
1 max_used_connections / max_connections * 100% = 3/100 * 100% ≈ 3%
%
```

查看最大连接数与响应最大连接数：

```
1 show variables like '%max_connections%';
2 show variables like '%max_user_connections%';
```

在配置文件 my.cnf 中修改最大连接数

```
1 [mysqld]
2 max_connections = 100
3 max_used_connections = 20
```

查询耗时0.5秒

建议将单次查询耗时控制在0.5秒以内，0.5秒是个经验值，源于用户体验的3秒原则。如果用户的操作3秒内没有响应，将会厌烦甚至退出。响应时间=客户端UI渲染耗时+网络请求耗时+应用程序处理耗时+查询数据库耗时，0.5秒就是留给数据库1/6的处理时间。

实施原则

相比NoSQL数据库，MySQL是个娇气脆弱的家伙。它就像体育课上的女同学，一点纠纷就和同学闹别扭(扩容难)，跑两步就气喘吁吁(容量小并发低)，常常身体不适要请假(SQL约束太多)。如今大家都会搞点分布式，应用程序扩容比数据库要容易得多，所以实施原则是**数据库少干活，应用程序多干活**。

- 充分利用但不滥用索引，须知索引也消耗磁盘和CPU。
- 不推荐使用数据库函数格式化数据，交给应用程序处理。
- 不推荐使用外键约束，用应用程序保证数据准确性。
- 写多读少的场景，不推荐使用唯一索引，用应用程序保证唯一性。
- 适当冗余字段，尝试创建中间表，用应用程序计算中间结果，用空间换时间。
- 不允许执行极度耗时的事务，配合应用程序拆分成更小的事务。
- 预估重要数据表（比如订单表）的负载和数据增长态势，提前优化。

数据表设计

数据类型

数据类型的选择原则：更简单或者占用空间更小。

- 如果长度能够满足，整型尽量使用tinyint、smallint、medium_int而非int。
- 如果字符串长度确定，采用char类型。
- 如果varchar能够满足，不采用text类型。
- 精度要求较高的使用decimal类型，也可以使用BIGINT，比如精确两位小数就乘以100后保存。
- 尽量采用timestamp而非datetime。

类型	占据字节	描述
datetime	8字节	'1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'
timestamp	4字节	'1970-01-01 00:00:01.000000' to '2038-01-19 03:14:07.999999'

相比datetime，timestamp占用更少的空间，以UTC的格式储存自动转换时区。

避免空值

MySQL中字段为NULL时依然占用空间，会使索引、索引统计更加复杂。从NULL值更新到非NULL无法做到原地更新，容易发生

索引分裂影响性能。尽可能将NULL值用有意义的值代替，也能避免SQL语句里面包含is not null的判断。微信搜索web_resource 关注获取更多推送。微信搜索web_resource 关注获取更多推送。

text类型优化

由于text字段储存大量数据，表容量会很早涨上去，影响其他字段的查询性能。建议抽取出来放在子表里，用业务主键关联。

索引优化

索引分类

- 普通索引：最基本的索引。
- 组合索引：多个字段上建立的索引，能够加速复合查询条件的检索。
- 唯一索引：与普通索引类似，但索引列的值必须唯一，允许有空值。
- 组合唯一索引：列值的组合必须唯一。
- 主键索引：特殊的唯一索引，用于唯一标识数据表中的某一条记录，不允许有空值，一般用primary key约束。
- 全文索引：用于海量文本的查询，MySQL5.6之后的InnoDB和MyISAM均支持全文索引。由于查询精度以及扩展性不佳，更多的企业选择Elasticsearch。

索引优化

- 分页查询很重要，如果查询数据量超过30%，MySQL不会使用索引。
- 单表索引数不超过5个、单个索引字段数不超过5个。
- 字符串可使用前缀索引，前缀长度控制在5-8个字符。
- 字段唯一性太低，增加索引没有意义，如：是否删除、性别。

合理使用覆盖索引，如下所示：

```
1 select login_name, nick_name from member where login_name = ?
```

login_name, nick_name两个字段建立组合索引，比login_name简单索引要更快。

SQL优化

分批处理

博主小时候看到鱼塘挖开小口子放水，水面有各种漂浮物。浮萍和树叶总能顺利通过出水口，而树枝会挡住其他物体通过，有时还会卡住，需要人工清理。MySQL就是鱼塘，最大并发数和网络带宽就是出水口，用户SQL就是漂浮物。微信搜索web_resource 关注获取更多推送。

不带分页参数的查询或者影响大量数据的update和delete操作，都是树枝，我们要把它打散分批处理，举例说明：

业务描述：更新用户所有已过期的优惠券为不可用状态。

SQL语句：

```
1 update status=0 FROM `coupon` WHERE expire_date <= #{currentDate} and status=1;
```

如果大量优惠券需要更新为不可用状态，执行这条SQL可能会堵死其他SQL，分批处理伪代码如下：

```
1 int pageNo = 1
2 ;
3 int PAGE_SIZE = 100
4 ;
5 while(true) {
6     List<Integer> batchIdList = queryList('select id FROM `coupon` WHERE expire_date <= #{current
7 );
8     if (CollectionUtils.isEmpty(batchIdList)) {
9         return
10    ;
11    }
12    update('update status = 0 FROM `coupon` where status = 1 and id in #{batchIdList}')
13    pageNo++;
14 }
```

操作符<>优化

通常<>操作符无法使用索引，举例如下，查询金额不为100元的订单：

```
1 select id from orders where amount != 100
2 ;
```

如果金额为100的订单极少，这种数据分布严重不均的情况下，有可能使用索引。鉴于这种不确定性，采用union聚合搜索结果，改写方法如下：

```
1 (select id from orders where amount > 100
2 )
3 union all
4 (select id from orders where amount < 100 and amount > 0
5 )
```

OR优化

在Innodb引擎下or无法使用组合索引，比如：

```
1 select id, product_name from orders where mobile_no = '13421800407' or user_id = 100
2 ;
```

OR无法命中mobile_no + user_id的组合索引，可采用union，如下所示：

```
1 (select id, product_name from orders where mobile_no = '13421800407'
2 )
3 union
4 (select id, product_name from orders where user_id = 100)
5 ;
```

此时id和product_name字段都有索引，查询才最高效。

IN优化

IN适合主表大子表小，EXIST适合主表小子表大。由于查询优化器的不断升级，很多场景这两者性能差不多一样了。

尝试改为join查询，举例如下：

```
1 select id from orders where user_id in (select id from user where level = 'VIP')
2 ;
```

采用JOIN如下所示：

```
1 select o.id from orders o left join user u on o.user_id = u.id where u.level = 'VIP'
2 ;
```

不做列运算

通常在查询条件列运算会导致索引失效，如下所示：

查询当日订单

```
1 select id from order where date_format(create_time, '%Y-%m-%d') = '2019-07-01'
2 ;
```

date_format函数会导致这个查询无法使用索引，改写后：

```
1 select id from order where create_time between '2019-07-01 00:00:00' and '2019-07-01 23:59:59'
2 ;
```

避免Select all

如果不查询表中所有的列，避免使用SELECT *，它会进行全表扫描，不能有效利用索引。

Like优化

like用于模糊查询，举个例子（field已建立索引）：

```
1 SELECT column FROM table WHERE field like '%keyword%';
```

这个查询未命中索引，换成下面的写法：

```
1 SELECT column FROM table WHERE field like 'keyword%';
```

去除了前面的%查询将会命中索引，但是产品经理一定要前后模糊匹配呢？全文索引fulltext可以尝试一下，但Elasticsearch才是终极武器。

Join优化

join的实现是采用Nested Loop Join算法，就是通过驱动表的结果集作为基础数据，通过该结数据作为过滤条件到下一个表中循环查询数据，然后合并结果。如果有多个join，则将前面的结果集作为循环数据，再次到后一个表中查询数据。

驱动表和被驱动表尽可能增加查询条件，满足ON的条件而少用Where，用小结果集驱动大结果集。

被驱动表的join字段上加上索引，无法建立索引的时候，设置足够的Join Buffer Size。

禁止join连接三个以上的表，尝试增加冗余字段。微信搜索web_resource 关注获取更多推送。

Limit优化

limit用于分页查询时越往后翻性能越差，解决的原则：缩小扫描范围，如下所示：

```
1 select * from orders order by id desc limit 1000000,10
```

耗时0.4秒

```
1 select * from orders order by id desc limit 1000000,1
```

```
0
```

耗时5.2秒

先筛选出ID缩小查询范围，写法如下：

```
1 select * from orders where id > (select id from orders order by id desc limit 1000000, 1) order by id desc
```

耗时0.5秒

如果查询条件仅有主键ID，写法如下：

```
1 select id from orders where id between 1000000 and 1000010 order by id desc
```

如果以上方案依然很慢呢？只好用游标了，感兴趣的朋友阅读JDBC使用游标实现分页查询的方法

其他数据库

作为一名后端开发人员，务必精通作为存储核心的MySQL或SQL Server，也要积极关注NoSQL数据库，他们已经足够成熟并被广泛采用，能解决特定场景下的性能瓶颈。微信搜索web_resource 关注获取更多推送。

分类	数据库	特性
键值型	Memcache	用于内容缓存，大量数据的高访问负载
键值型	Redis	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	HBase	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	MongoDb	知名文档型数据库，也可以用于缓存
文档型	CouchDB	Apache的开源项目，专注于易用性，支持REST API
文档型	SequoiaDB	国内知名文档型数据库
图形	Neo4J	用于社交网络构建关系图谱，推荐系统等

参考

[1] <https://www.jianshu.com/p/6864abb4d885>

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. [Java后端优质文章整理](#)
2. [IDEA 远程一键部署 Spring Boot 到 Docker](#)

- [3. 这 26 条,你赞同几个?](#)
- [4. 7 个开源的 Spring Boot 前后端分离项目](#)
- [5. 如何设计 API 接口,实现统一格式返回?](#)



Java后端

长按识别二维码, 关注我的公众号

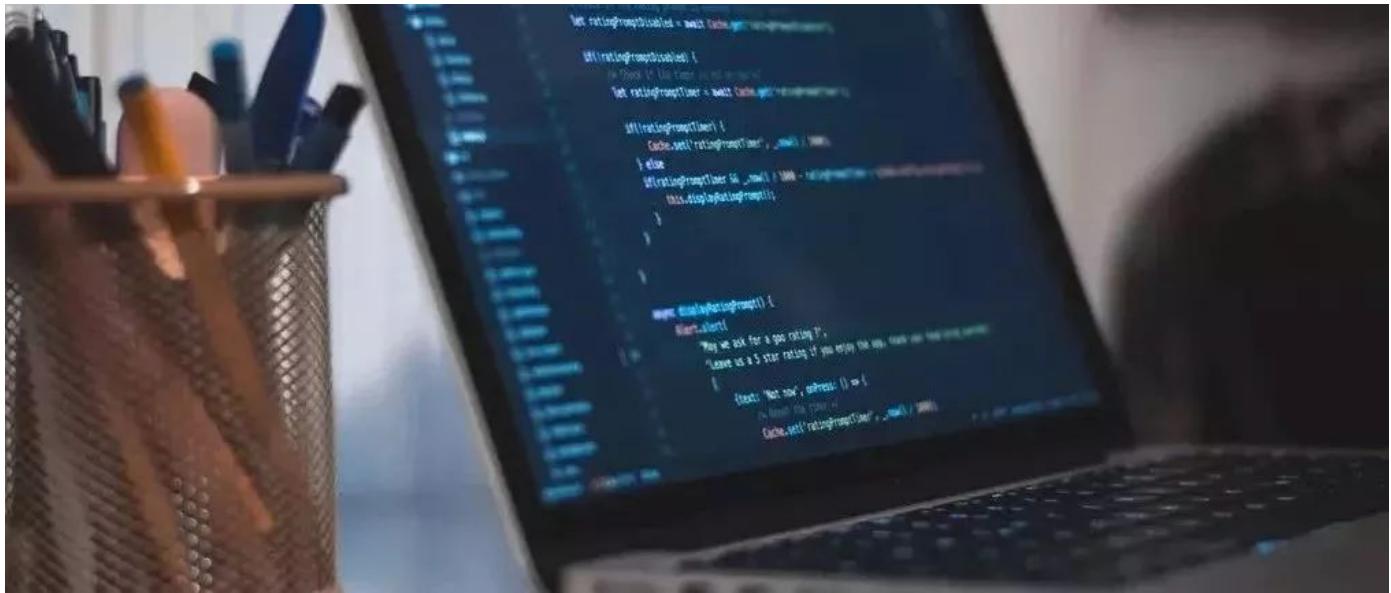
喜欢文章, 点个在看 

[阅读原文](#)

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

如何干掉恶心的 SQL 注入？

Java后端 1月20日



来源: b1ngz.github.io/java-sql-injection-note/

简介

文章主要内容包括：

- Java 持久层技术/框架简单介绍
- 不同场景/框架下易导致 SQL 注入的写法
- 如何避免和修复 SQL 注入

JDBC

介绍

- 全称 Java Database Connectivity
- 是 Java 访问数据库的 API，不依赖于特定数据库 (database-independent)
- 所有 Java 持久层技术都基于 JDBC

说明

直接使用 JDBC 的场景，如果代码中存在拼接 SQL 语句，那么很有可能会产生注入，如

```
// concat sql
String sql = "SELECT * FROM users WHERE name ='" + name + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

安全的写法是使用 **参数化查询 (parameterized queries)**，即 SQL 语句中使用参数绑定(? 占位符) 和 PreparedStatement，如

```
// use ? to bind variables
String sql = "SELECT * FROM users WHERE name= ? ";
PreparedStatement ps = connection.prepareStatement(sql);
// 参数 index 从 1 开始
ps.setString(1, name);
```

还有一些情况，比如 order by、column name，不能使用参数绑定，此时需要手工过滤，如通常 order by 的字段名是有限的，因此可以使用白名单的方式来限制参数值

这里需要注意的是，使用了 PreparedStatement **并不意味着不会产生注入**，如果在使用 PreparedStatement 之前，存在拼接 sql 语句，

那么仍然会导致注入，如

```
// 拼接 sql  
String sql = "SELECT * FROM users WHERE name ='" + name + "'";  
PreparedStatement ps = connection.prepareStatement(sql);
```

看到这里，大家肯定会好奇 PreparedStatement 是如何防止 SQL 注入的，来了解一下

正常情况下，用户的输入是作为参数值的，而在 SQL 注入中，用户的输入是作为 SQL 指令的一部分，会被数据库进行编译/解释执行。

当使用了 PreparedStatement，带占位符 (?) 的 sql 语句只会被编译一次，之后执行只是将占位符替换为用户输入，并不会再次编译/解释，因此从根本上防止了 SQL 注入问题。

Mybatis

介绍

- 首个 class persistence framework
- 介于 JDBC (raw SQL) 和 Hibernate (ORM)
- 简化绝大部分 JDBC 代码、手工设置参数和获取结果
- 灵活，使用者能够完全控制 SQL，支持高级映射

更多请参考: <http://www.mybatis.org>

说明

在 MyBatis 中，使用 XML 文件 或 Annotation 来进行配置和映射，将 interfaces 和 Java POJOs (Plain Old Java Objects) 映射到 database records。

XML 例子

Mapper Interface

```
@Mapper  
public interface UserMapper {  
    User getById(int id);  
}
```

XML 配置文件

```
<select id="getById" resultType="org.example.User">  
    SELECT * FROM user WHERE id = #{id}  
</select>
```

Annotation 例子

```
@Mapper  
public interface UserMapper {  
    @Select("SELECT * FROM user WHERE id= #{id}")  
    User getById(@Param("id") int id);  
}
```

可以看到，使用者需要自己编写 SQL 语句，因此当使用不当时，会导致注入问题与使用 JDBC 不同的是，MyBatis 使用 #{} 和 \${} 来进行参数值替换。

使用 #{} 语法时，MyBatis 会自动生成 PreparedStatement，使用参数绑定 (?) 的方式来设置值，上述两个例子等价的 JDBC 查询代码如下：

```
String sql = "SELECT * FROM users WHERE id = ?";  
PreparedStatement ps = connection.prepareStatement(sql);  
ps.setInt(1, id);
```

因此 #{} 可以有效防止 SQL 注入，详细可参考 <http://www.mybatis.org/mybatis-3/sqlmap-xml.html> **String Substitution** 部分。

而使用 \${} 语法时，MyBatis 会直接注入原始字符串，即相当于拼接字符串，因而会导致 SQL 注入，如

```
<select id="getByName" resultType="org.example.User">  
    SELECT * FROM user WHERE name = '${name}' limit 1  
</select>
```

' or

name 值为 '1'='1' ，实际执行的语句为

```
SELECT * FROM user WHERE name = '' or '1'='1' limit 1
```

因此建议尽量使用 #{}，但有些时候，如 order by 语句，使用 #{} 会导致出错，如

```
ORDER BY #{sortBy}
```

sortBy 参数值为 name ，替换后会成为

```
ORDER BY "name"
```

即以字符串 “name” 来排序，而非按照 name 字段排序

详细可参考: <https://stackoverflow.com/a/32996866/6467552>

这种情况就需要使用 \${}

```
ORDER BY ${sortBy}
```

使用了 \${} 后，使用者需要自行过滤输入，方法有：

代码层使用白名单的方式，限制 sortBy 允许的值，如只能为 name , email 字段，异常情况则设置为默认值 name

在 XML 配置文件中，使用 if 标签来进行判断

Mapper 接口方法

```
List<User> getUserListSortBy(@Param("sortBy") String sortBy);
```

xml 配置文件

```
<select id="getUserListSortBy" resultType="org.example.User">  
    SELECT * FROM user  
    <if test="sortBy == 'name' or sortBy == 'email'">  
        order by ${sortBy}  
    </if>  
</select>
```

因为 Mybatis 不支持 else，需要默认值的情况，可以使用 choose(when,otherwise)

```
<select id="getUserListSortBy" resultType="org.example.User">
    SELECT * FROM user
    <choose>
        <when test="sortBy == 'name' or sortBy == 'email'">
            order by ${sortBy}
        </when>
        <otherwise>
            order by name
        </otherwise>
    </choose>
</select>
```

更多场景

除了 orderby 之外，还有一些可能会使用到 \${} 情况，可以使用其他方法避免，如

like 语句

- 如需要使用通配符 (wildcard characters % 和 _)，可以
- 在代码层，在参数值两边加上 %，然后再使用 #{}
- 使用 bind 标签来构造新参数，然后再使用 #{}

Mapper 接口方法

```
List<User> getUserListLike(@Param("name") String name);
```

xml 配置文件

```
<select id="getUserListLike" resultType="org.example.User">
    <bind name="pattern" value="%${name}%" />
    SELECT * FROM user
    WHERE name LIKE #{pattern}
</select>
```

<bind> 语句内的 value 为 OGNL expression

具体可参考：

<http://www.mybatis.org/mybatis-3/dynamic-sql.html>

bind 部分使用 SQL concat() 函数

```
<select id="getUserListLikeConcat" resultType="org.example.User">
    SELECT * FROM user WHERE name LIKE concat ('%', #{name}, '%')
</select>
```

除了注入问题之外，这里还需要对用户的输入进行过滤，不允许有通配符，否则在表中数据量较多的时候，假设用户输入为 %%，会进行全表模糊查询，严重情况下可导致 DOS

参考：

<http://www.tothenew.com/blog/sql-wildcards-is-your-application-safe>

IN 条件

- 使用 <foreach> 和 #{}
- Mapper 接口方法

```
List<User> getUserListIn(@Param("nameList") List<String> nameList);
```

xml 配置文件

```
<select id="selectUserIn" resultType="com.example.User">
    SELECT * FROM user WHERE name in
    <foreach item="name" collection="nameList"
        open="(" separator="," close=")"
        #{name}
    </foreach>
</select>
```

具体可参考

<http://www.mybatis.org/mybatis-3/dynamic-sql.html>

foreach 部分

limit 语句

- 直接使用 #{} 即可
- Mapper 接口方法

```
List<User> getUserListLimit(@Param("offset") int offset, @Param("limit") int limit);
```

xml 配置文件

```
<select id="getUserListLimit" resultType="org.example.User">
    SELECT * FROM user limit #{offset}, #{limit}
</select>
```

JPA & Hibernate

介绍

JPA:

- 全称 Java Persistence API
- ORM (object-relational mapping) 持久层 API，需要有具体的实现

更多请参考：

<https://en.wikipedia.org/wiki/JavaPersistenceAPI>

Hibernate:

- JPA ORM 实现

更多请参考 <http://hibernate.org>。

说明

这里有一种错误的认识，使用了 ORM 框架，就不会有 SQL 注入。而实际上，在 Hibernate 中，支持 HQL (Hibernate Query Language) 和 native sql 查询，前者存在 HQL 注入，后者和之前 JDBC 存在相同的注入问题，来具体看一下。

HQL

HQL 查询例子

```
Query<User> query = session.createQuery("from User where name = '" + name + "'", User.class);
User user = query.getSingleResult();
```

这里的 User 为类名，和原生 SQL 类似，拼接会导致注入。

正确的用法：

- 位置参数 (Positional parameter)

```
Query<User> query = session.createQuery("from User where name =?", User.class);
query.setParameter(0, name);
```

- 命名参数 (named parameter)

```
Query<User> query = session.createQuery("from User where name = :name", User.class);
query.setParameter("name", name);
```

- 命名参数 list (named parameter list)

```
Query<User> query = session.createQuery("from User where name in (:nameList)", User.class);
query.setParameterList("nameList", Arrays.asList("lisi", "zhaowu"));
```

- 类实例 (JavaBean)

```
User user = new User();
user.setName("zhaowu");
Query<User> query = session.createQuery("from User where name = :name", User.class);
// User 类需要有 getName() 方法
query.setProperties(user);
```

Native SQL

存在 SQL 注入

```
String sql = "select * from user where name = '" + name + "'";
// deprecated
// Query query = session.createSQLQuery(sql);
Query query = session.createNativeQuery(sql);
```

使用参数绑定来设置参数值

```
String sql = "select * from user where name = :name";
// deprecated
// Query query = session.createSQLQuery(sql);
Query query = session.createNativeQuery(sql);
query.setParameter("name", name);
```

JPA

JPA 中使用 JPQL (Java Persistence Query Language)，同时也支持 native sql，因此和 Hibernate 存在类似的问题，这里就不再细说，感兴趣的可以参考：

<https://software-security.sans.org/developer-how-to/fix-sql-injection-in-java-persistence-api-jpa>



微信扫描二维码，关注我的公众号

少侠！如何写一手好 SQL？

编码砖家 Java后端 3月6日

Java后端

博文、教程、干货、资讯

博主（编码砖家）负责的项目主要采用阿

里云数据库MySQL，最近频繁出现慢SQL告警，**执行时间最长的竟然高达5分钟**。导出日志后分析，主要原因竟然是**没有命中索引和没有分页处理**。

其实这是非常低级的错误，我不禁后背一凉，团队成员的技术水平亟待提高啊。改造这些SQL的过程中，总结了一些经验分享给大家，如果有错误欢迎批评指正。

MySQL性能

最大数据量

抛开数据量和并发数，谈性能都是耍流氓。MySQL没有限制单表最大记录数，它取决于操作系统对文件大小的限制。

文件系统	单文件大小限制
FAT32	最大4G
NTFS	最大64GB
NTFS5.0	最大2TB
EXT2	块大小为1024字节，文件最大容量16GB；块大小为4096字节，文件最大容量2TB
EXT3	块大小为4KB，文件最大容量为4TB
EXT4	理论可以大于16TB

《阿里巴巴Java开发手册》提出单表行数超过500万行或者单表容量超过2GB，才推荐分库分表。性能由综合因素决定，抛开业务复杂度，影响程度依次是硬件配置、MySQL配置、数据表设计、索引优化。500万这个值仅供参考，并非铁律。博主曾经操作过超过4亿行数据的单表，分页查询最新的20条记录耗时0.6秒，SQL语句大致是 `select field_1,field_2 from table where id < # {prePageMinId} order by id desc limit 20`，`prePageMinId` 是上一页数据记录的最小ID。虽然当时查询速度还凑合，随着数据不断增长，有朝一日必定不堪重负。分库分表是个周期长而风险高的大活儿，应该尽可能在当前结构上优化，比如升级硬件、迁移历史数据等等，实在没辙了再分。对分库分表感兴趣的同学可以阅读分库分表的基本思想。

最大并发数

并发数是指同一时刻数据库能处理多少个请求，由`max_connections`和`max_user_connections`决定。****maxconnections是指MySQL实例的最大连接数，上限值是16384，maxuserconnections是指每个数据库用户的最大连接数。MySQL会为每个连接提供缓冲区，意味着消耗更多的内存。如果连接数设置太高硬件吃不消，太低又不能充分利用硬件。一般要求两者比值超过10%，计算方法如下：**

```
max_used_connections / max_connections * 100% = 3/100 *100% ≈ 3%
%
```

查看最大连接数与响应最大连接数：

```
show variables like '%max_connections%';show variables like '%max_user_connections%'  
;
```

在配置文件my.cnf中修改最大连接数

```
[mysqld]max_connections = 100max_used_connections = 2  
0
```

查询耗时0.5秒

建议将单次查询耗时控制在0.5秒以内，0.5秒是个经验值，源于用户体验的 **3秒原则**。如果用户的操作3秒内没有响应，将会厌烦甚至退出。响应时间=客户端UI渲染耗时+网络请求耗时+应用程序处理耗时+查询数据库耗时，0.5秒就是留给数据库1/6的处理时间。

实施原则

相比NoSQL数据库，MySQL是个娇气脆弱的家伙。它就像体育课上的女同学，一点纠纷就和同学闹别扭(扩容难)，跑两步就气喘吁吁(容量小并发低)，常常身体不适要请假(SQL约束太多)。如今大家都会搞点分布式，应用程序扩容比数据库要容易得多，所以实施原则是 **数据库少干活，应用程序多干活**。

- 充分利用但不滥用索引，须知索引也消耗磁盘和CPU。
- 不推荐使用数据库函数格式化数据，交给应用程序处理。
- 不推荐使用外键约束，用应用程序保证数据准确性。
- 写多读少的场景，不推荐使用唯一索引，用应用程序保证唯一性。
- 适当冗余字段，尝试创建中间表，用应用程序计算中间结果，用空间换时间。
- 不允许执行极度耗时的事务，配合应用程序拆分成更小的事务。
- 预估重要数据表（比如订单表）的负载和数据增长态势，提前优化。

数据表设计

数据类型

数据类型的选择原则：更简单或者占用空间更小。

- 如果长度能够满足，整型尽量使用tinyint、smallint、medium_int而非int。
- 如果字符串长度确定，采用char类型。
- 如果varchar能够满足，不采用text类型。
- 精度要求较高的使用decimal类型，也可以使用BIGINT，比如精确两位小数就乘以100后保存。

尽量采用timestamp而非datetime。

类型	占据字节	描述
----	------	----

类型	占据字节	描述
datetime	8字节	'1000-01-01 00:00:00.000000' to '9999-12-31 23:59:59.999999'
timestamp	4字节	'1970-01-01 00:00:01.000000' to '2038-01-19 03:14:07.999999'

相比datetime, timestamp占用更少的空间，以UTC的格式储存自动转换时区。

避免空值

MySQL中字段为NULL时依然占用空间，会使索引、索引统计更加复杂。从NULL值更新到非NULL无法做到原地更新，容易发生索引分裂影响性能。尽可能将NULL值用有意义的值代替，也能避免SQL语句里面包含 is not null的判断。

text类型优化

由于text字段储存大量数据，表容量会很早涨上去，影响其他字段的查询性能。建议抽取出来放在子表里，用业务主键关联。

索引优化

索引分类

1. 普通索引：最基本的索引。
2. 组合索引：多个字段上建立的索引，能够加速复合查询条件的检索。
3. 唯一索引：与普通索引类似，但索引列的值必须唯一，允许有空值。
4. 组合唯一索引：列值的组合必须唯一。
5. 主键索引：特殊的唯一索引，用于唯一标识数据表中的某一条记录，不允许有空值，一般用primary key约束。
6. 全文索引：用于海量文本的查询，MySQL5.6之后的InnoDB和MyISAM均支持全文索引。由于查询精度以及扩展性不佳，更多的企业选择Elasticsearch。

索引优化

1. 分页查询很重要，如果查询数据量超过30%，MySQL不会使用索引。
2. 单表索引数不超过5个、单个索引字段数不超过5个。
3. 字符串可使用前缀索引，前缀长度控制在5-8个字符。
4. 字段唯一性太低，增加索引没有意义，如：是否删除、性别。
5. 合理使用覆盖索引，如下所示：

```
select loginname, nickname from member where login_name = ?
```

loginname, nickname两个字段建立组合索引，比login_name简单索引要更快

SQL优化

分批处理

博主小时候看到鱼塘挖开小口子放水，水面有各种漂浮物。浮萍和树叶总能顺利通过出水口，而树枝会挡住其他物体通过，有时还会卡住，需要人工清理。MySQL就是鱼塘，最大并发数和网络带宽就是出水口，用户SQL就是漂浮物。不带分页参数的查询或者影响大量数据的update和delete操作，都是树枝，我们要把它打散分批处理，举例说明：业务描述：更新用户所有已过期的优惠券为不可用状态。SQL语句：update status=0 FROM coupon WHERE expire_date <= #{currentDate} and status=1;如果大量优惠券需要更新为不可用状态，执行这条SQL可能会堵死其他SQL，分批处理伪代码如下：

```
int pageNo = 1;
int PAGE_SIZE = 100;
while(true) {
    List<Integer> batchIdList = queryList('select id FROM `coupon` WHERE expire_date <= #{currentDate} and status = 1 limit #{(pageNo-1)*PAGE_SIZE,PAGE_SIZE}');
    if(CollectionUtils.isEmpty(batchIdList)) {
        return;
    }
    update('update status = 0 FROM `coupon` where status = 1 and id in #{batchIdList}');
    pageNo++;
}
```

操作符<>优化

通常<>操作符无法使用索引，举例如下，查询金额不为100元的订单：select id from orders where amount != 100;如果金额为100的订单极少，这种数据分布严重不均的情况下，有可能使用索引。鉴于这种不确定性，采用union聚合搜索结果，改写方法如下：

```
(select id from orders where amount > 100) union all(select id from orders where amount < 100 and amount > 0)
```

OR优化

在Innodb引擎下or无法使用组合索引，比如：

```
select id, product_name from orders where mobile_no = '13421800407' or user_id = 100;
```

OR无法命中mobile_no + user_id的组合索引，可采用union，如下所示：

```
(select id, product_name from orders where mobile_no = '13421800407') union(select id, product_name from orders where user_id = 100)
```

此时id和product_name字段都有索引，查询才最高效。

IN优化

1. IN适合主表大子表小，EXIST适合主表小子表大。由于查询优化器的不断升级，很多场景这两者性能差不多一样了。

2. 尝试改为join查询，举例如下：

```
select id from orders where user_id in (select id from user where level = 'VIP');
```

采用JOIN如下所示：

```
select o.id from orders o left join user u on o.user_id = u.id where u.level = 'VIP';
```

不做列运算

通常在查询条件列运算会导致索引失效，如下所示：查询当日订单

```
select id from order where date_format(create_time, '%Y-%m-%d') = '2019-07-01';
```

date_format函数会导致这个查询无法使用索引，改写后：

```
select id from order where create_time between '2019-07-01 00:00:00' and '2019-07-01 23:59:59';
```

避免Select all

如果不查询表中所有的列，避免使用 SELECT *，它会进行全表扫描，不能有效利用索引。

Like优化

like用于模糊查询，举个例子（field已建立索引）：

```
SELECT column FROM table WHERE field like '%keyword%';
```

这个查询未命中索引，换成下面的写法：

```
SELECT column FROM table WHERE field like 'keyword%';
```

去除了前面的%查询将会命中索引，但是产品经理一定要前后模糊匹配呢？全文索引fulltext可以尝试一下，但Elasticsearch才是终极武器。

Join优化

join的实现是采用Nested Loop Join算法，就是通过驱动表的结果集作为基础数据，通过该结数据作为过滤条件到下一个表中循环查询数据，然后合并结果。如果有多个join，则将前面的结果集作为循环数据，再次到后一个表中查询数据。

1. 驱动表和被驱动表尽可能增加查询条件，满足ON的条件而少用Where，用小结果集驱动大结果集。
2. 被驱动表的join字段上加上索引，无法建立索引的时候，设置足够的Join Buffer Size。
3. 禁止join连接三个以上的表，尝试增加冗余字段。

limit用于分页查询时越往后翻性能越差，解决的原则：缩小扫描范围，如下所示：

```
select * from orders order by id desc limit 100000,10 耗时0.4秒
select * from orders order by id desc limit 1000000,10耗时5.2秒
```

先筛选出ID缩小查询范围，写法如下：

```
select * from orders where id > (select id from orders order by id desc limit 1000000, 1) order by id desc limit 0,10耗时0.5秒
```

如果查询条件仅有主键ID，写法如下：

```
select id from orders where id between 1000000 and 1000010 order by id desc耗时0.3秒
```

如果以上方案依然很慢呢？只好用游标了，感兴趣的朋友阅读JDBC使用游标实现分页查询的方法

其他数据库

作为一名后端开发人员，务必精通作为存储核心的MySQL或SQL Server，也要积极关注NoSQL数据库，他们已经足够成熟并被广泛采用，能解决特定场景下的性能瓶颈。

分类	数据库	特性
键值型	Memcache	用于内容缓存，大量数据的高访问负载
键值型	Redis	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	HBase	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	MongoDb	知名文档型数据库，也可以用于缓存
文档型	CouchDB	Apache的开源项目，专注于易用性，支持REST API
文档型	SequoiaDB	国内知名文档型数据库
图形	Neo4J	用于社交网络构建关系图谱，推荐系统等

来源 | 编码砖家
[链接 | cnblogs.com/xiaoyangjia/p/11267191.html](http://cnblogs.com/xiaoyangjia/p/11267191.html)

- E N D -

推荐阅读

1. 如何从 Windows 切换到 Linux
2. GitHub 近 70K 星，命令行的艺术！
3. 一线大厂的分布式唯一ID生成方案
4. 探讨：为什么大公司要使用微服务？



微信搜一搜

Q Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

干货！MySQL 数据库开发规范

在云端 Java后端 2019-11-03

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

[上篇 | 35 个小细节, 提升 Java 代码运行效率](#)

1. 所有的数据库对象名称必须使用小写字母并用下划线分割 (MySQL大小写敏感, 名称要见名知意, 最好不超过32字符)
2. 所有的数据库对象名称禁止使用MySQL保留关键字 (如 desc、range、match、delayed 等, 请参考 MySQL官方保留字)
3. 临时库表必须以tmp为前缀并以日期为后缀 (tmp_)
4. 备份库和库必须以bak为前缀并以日期为后缀(bak_)
- 5.所有存储相同数据的列名和列类型必须一致。 (在多个表中的字段如user_id, 它们类型必须一致)
6. mysql5.5之前默认的存储的引擎是myisam, 没有特殊要求, 所有的表必须使用innodb (innodb好处支持失误, 行级锁, 高并发下性能更好, 对多核, 大内存, ssd等硬件支持更好)
7. 数据库和表的字符集尽量统一使用utf8 (字符集必须统一, 避免由于字符集转换产生的乱码, 汉字utf8下占3个字节)
8. 所有表和字段都要添加注释COMMENT, 从一开始就进行数据字典的维护
9. 尽量控制单表数据量的大小在500w以内, 超过500w可以使用历史数据归档, 分库分表来实现 (500万行并不是MySQL数据库的限制。过大对于修改表结构, 备份, 恢复都会有很大问题。MySQL没有对存储有限制, 取决于存储设置和文件系统)
10. 谨慎使用mysql分区表 (分区表在物理上表现为多个文件, 在逻辑上表现为一个表)
11. 谨慎选择分区键, 跨分区查询效率可能更低
12. 建议使用物理分表的方式管理大数据
13. 尽量做到冷热数据分离, 减小表的宽度 (mysql限制最多存储4096列, 行数没有限制, 但是每一行的字节总数不能超过65535。列限制好处: 减少磁盘io, 保证热数据的内存缓存命中率, 避免读入无用的冷数据)
14. 禁止在表中建立预留字段 (无法确认存储的数据类型, 对预留字段类型进行修改, 会对表进行锁定)
15. 禁止在数据中存储图片, 文件二进制数据 (使用文件服务器)
16. 禁止在线上做数据库压力测试
17. 禁止从开发环境, 测试环境直接连生产环境数据库
18. 限制每张表上的索引数量, 建议单表索引不超过5个 (索引会增加查询效率, 但是会降低插入和更新的速度)
19. 避免建立冗余索引和重复索引 (冗余: index (a,b,c) index(a,b) index(a))

20. 禁止给表中的每一列都建立单独的索引

21. 每个innodb表必须有一个主键，选择自增id（不能使用更新频繁的列作为主键，不适用UUID,MD5,HASH,字符串列作为主键）

22. 区分度最高的列放在联合索引的最左侧

23. 尽量把字段长度小的列放在联合索引的最左侧

24. 尽量避免使用外键（禁止使用物理外键，建议使用逻辑外键）

25. 优先选择符合存储需要的最小数据类型

26. 优先使用无符号的整形来存储

27. 优先选择存储最小的数据类型（varchar(N),N代表的是字符数，而不是字节数，N代表能存储多少个汉字）

28. 避免使用Text或是Blob类型

29. 避免使用ENUM数据类型（修改ENUM值需要使用ALTER语句，ENUM类型的ORDER BY操作效率低，需要额外操作，禁止使用书值作为ENUM的枚举值

30. 尽量把所有的字段定义为NOT NULL（索引NULL需要额外的空间来保存，所以需要暂用更多的内存，进行比较和计算要对NULL值做特别的处理）

31. 使用timestamp或datetime类型来存储时间

32. 同财务相关的金额数据，采用decimal类型（不丢失精度，禁止使用float和double）

33. 避免使用双%号和like，搜索严禁左模糊或者全模糊（如果需要请用搜索引擎来解决。索引文件具有B-Tree的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引）

34. 建议使用预编译语句进行数据库操作

35. 禁止跨库查询（为数据迁移和分库分表留出余地，降低耦合度，降低风险）

36. 禁止select * 查询（消耗更多的cpu和io及网络带宽资源，无法使用覆盖索引）

37. 禁止使用不含字段列表的insert语句（不允许insert into t values ('a', 'b', 'c') 不允许）

38. in 操作能避免则避免，若实在避免不了，需要仔细评估in后边的集合元素数量，控制在1000个之内

39. 禁止使用order by rand() 进行随机排序

40. 禁止where从句中对列进行函数转换和计算（例如：where date(createtime) = '20160901' 会无法使用createtime列上索引。改成 where createtime>='20160901' and createtime <'20160902'）

41. 尽量使用 union all 代替 union

42. 拆分复杂的大SQL为多个小SQL（MySQL一个SQL只能使用一个CPU进行计算）

43. 尽量避免使用子查询，可以把子查询优化为join操作（子查询的结果集无法使用索引，子查询会产生临时表操作，如果子查询数据量大会影响效率，消耗过多的CPU及IO资源）

44. 超过100万行的批量写操作，要分批多次进行操作（大批量操作可能会造成严重的主从延迟，binlog日志为row格式会产生大量的日志，避免产生大事务操作）

45. 对于大表使用pt-online-schema-change修改表结构（避免大表修改产生的主从延迟，避免在对表字段进行修改时进行锁表）

46. 对于程序连接数据库账号，遵循权限最小原则

47. 超过三个表禁止 join。（需要 join 的字段，数据类型必须绝对一致；多表关联查询时，保证被关联的字段需要有索引。即使双表 join 也要注意表索引、SQL 性能。）

48. 在varchar字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度即可。

49. SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，如果可以是 consts 最好

50. 使用 ISNULL() 来判断是否为 NULL 值。

51. 尽量不要使用物理删除（即直接删除，如果要删除的话提前做好备份），而是使用逻辑删除，使用字段 delete_flag 做逻辑删除，类型为tinyint，0表示未删除，1表示已删除

52. 如果有 order by 的场景，请注意利用索引的有序性。order by 最后的字段是组合索引的一部分，并且放在索引组合顺序的最后，避免出现 file_sort 的情况，影响查询性能。

53. 在代码中写分页查询逻辑时，若 count 为 0 应直接返回，避免执行后面的分页语句

参考：

[1] 《阿里巴巴Java开发手册》

[2] 《高性能可扩展MySQL数据库设计及架构优化 电商项目》

链接：

作者 | 在云端

链接 | juejin.im/post/5c15c2b3f265da6170070613

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. 盘点阿里巴巴 33 个牛逼的开源项目
2. 为什么我不建议你去外包公司？
3. 理解 IntelliJ IDEA 的项目配置和 Web 部署
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践



学Java, 请关注公众号:Java后端

喜欢文章, 点个在看

干货！图解 MySQL 索引

浪人 Java后端 2019-10-07

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 狼人

链接 | cnblogs.com/liqiangchn/p/9060521.html

看了很多关于索引的博客，讲的大同小异。但是始终没有让我明白关于索引的一些概念，如B-Tree索引，Hash索引，唯一索引....或许有很多人和我一样，没搞清楚概念就开始研究B-Tree，B+Tree等结构，导致在面试的时候答非所问！

索引是什么？

索引是帮助MySQL高效获取数据的数据结构。

索引能干什么？

提高数据查询的效率。

索引：排好序的快速查找数据结构！索引会影响where后面的查找，和order by 后面的排序。

一、索引的分类

①从存储结构上来划分：BTREE索引（B-Tree或B+Tree索引），HASH索引，full-index全文索引，R-Tree索引。

②从应用层次来分：普通索引，唯一索引，复合索引

③根据中数据的物理顺序与键值的逻辑（索引）顺序关系：聚集索引，非聚集索引。

①中所描述的是索引存储时保存的形式，②是索引使用过程中进行的分类，两者是不同层次上的划分。不过平时讲的索引类型一般是指在应用层次的划分。

就像手机分类：安卓手机，IOS手机与 华为手机，苹果手机，OPPO手机一样。

普通索引：即一个索引只包含单个列，一个表可以有多个单列索引

唯一索引：索引列的值必须唯一，但允许有空值

复合索引：即一个索引包含多个列

聚簇索引(聚集索引)：并不是一种单独的索引类型，而是一种数据存储方式。具体细节取决于不同的实现，InnoDB的聚簇索引其实就是在同一个结构中保存了B-Tree索引(技术上来说是B+Tree)和数据行。

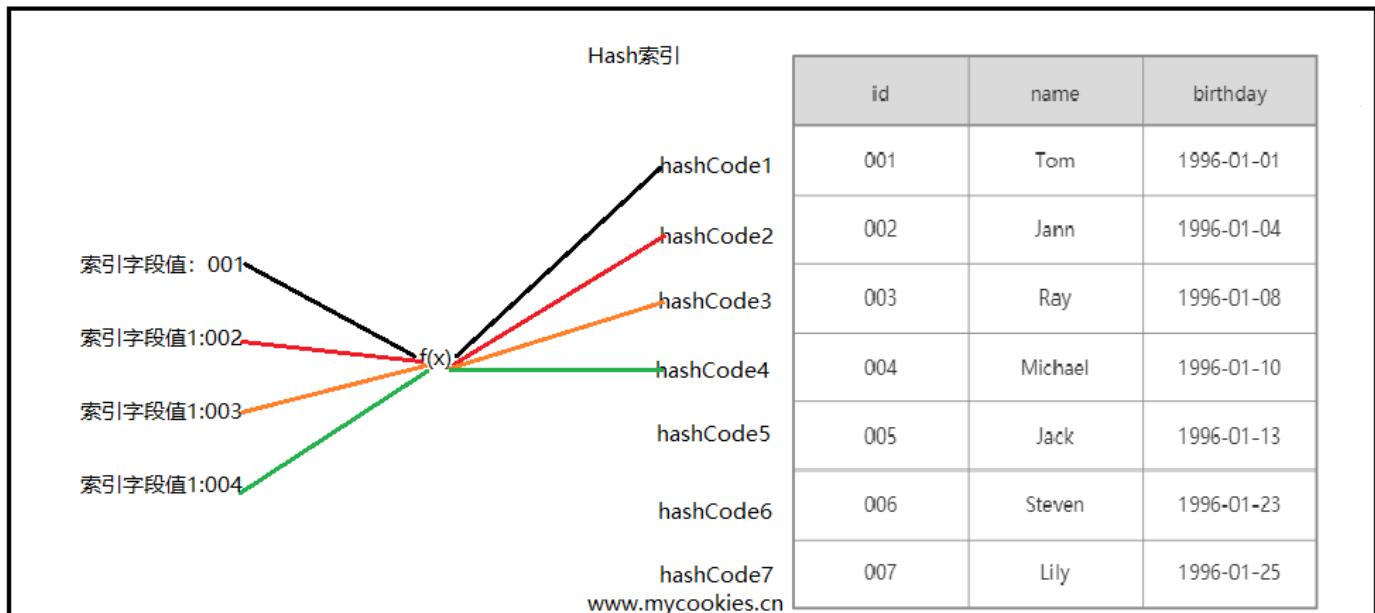
非聚簇索引：不是聚簇索引，就是非聚簇索引（认真脸）。

二、索引的底层实现

mysql默认存储引擎innodb只显式支持B-Tree(从技术上来说是B+Tree)索引，对于频繁访问的表，innodb会透明建立自适应hash索引，即在B树索引基础上建立hash索引，可以显著提高查找效率，对于客户端是透明的，不可控制的，隐式的。

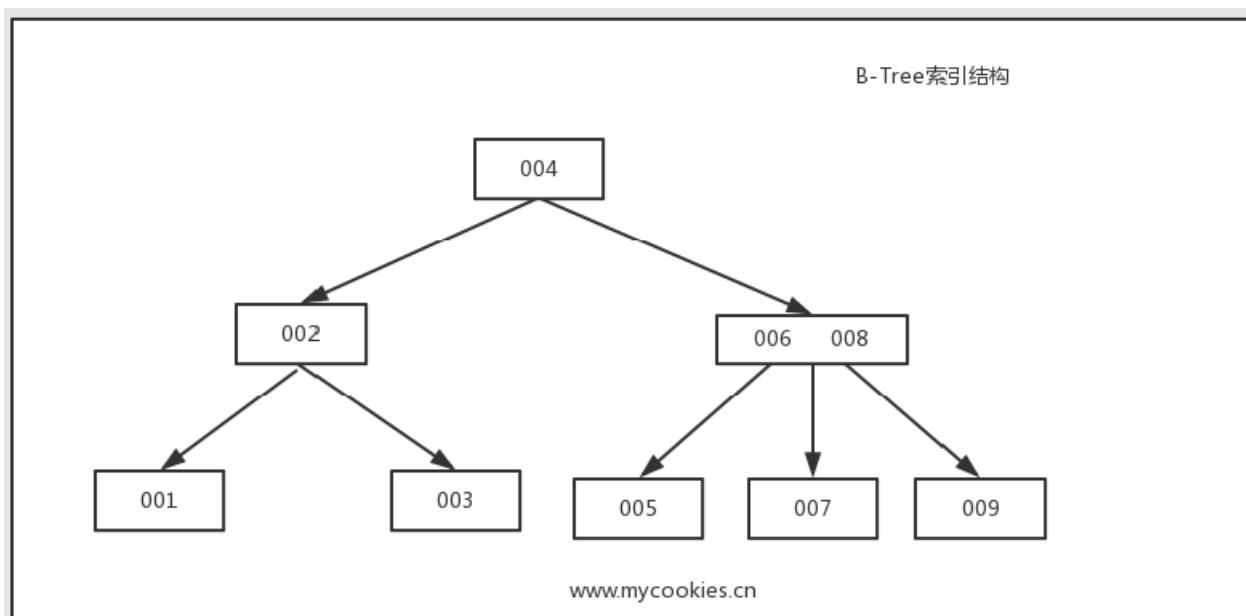
Hash索引

基于哈希表实现，只有精确匹配索引所有列的查询才有效，对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码（hash code），并且Hash索引将所有的哈希码存储在索引中，同时在索引表中保存指向每个数据行的指针。



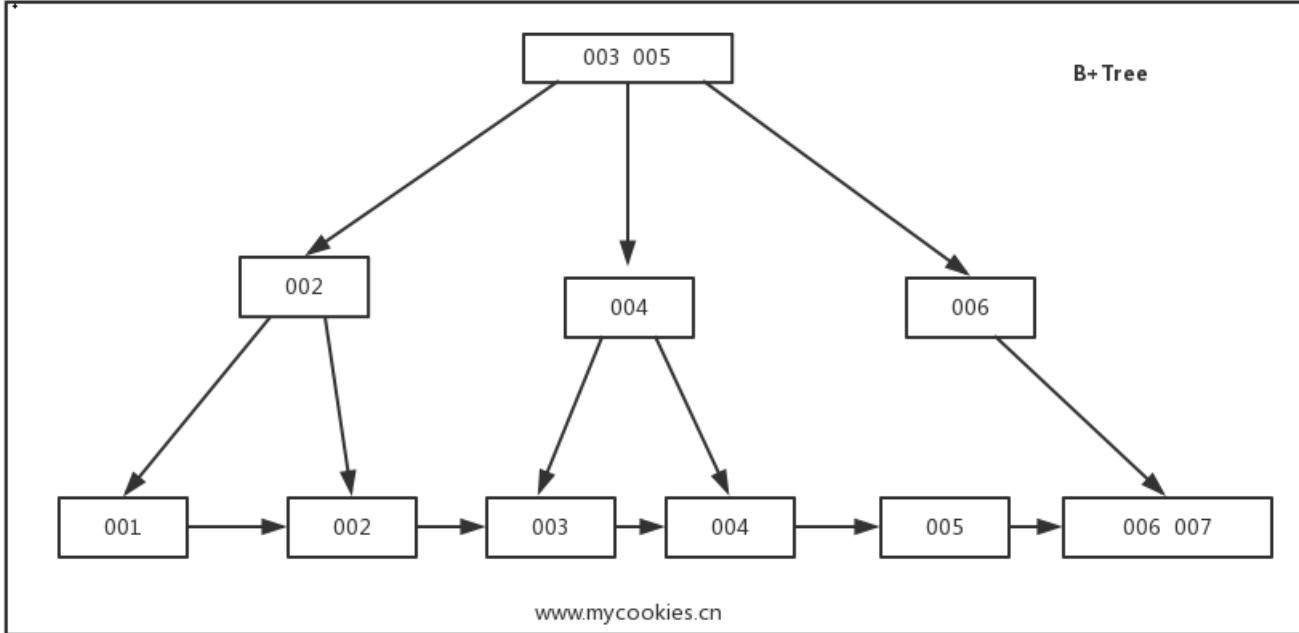
B-Tree索引 (MySQL使用B+Tree)

B-Tree能加快数据的访问速度，因为存储引擎不再需要进行全表扫描来获取数据，数据分布在各个节点之中。



B+Tree索引

是B-Tree的改进版本，同时也是数据库索引索引所采用的存储结构。数据都在叶子节点上，并且增加了顺序访问指针，每个叶子节点都指向相邻的叶子节点的地址。相比B-Tree来说，进行范围查找时只需要查找两个节点，进行遍历即可。而B-Tree需要获取所有节点，相比之下B+Tree效率更高。

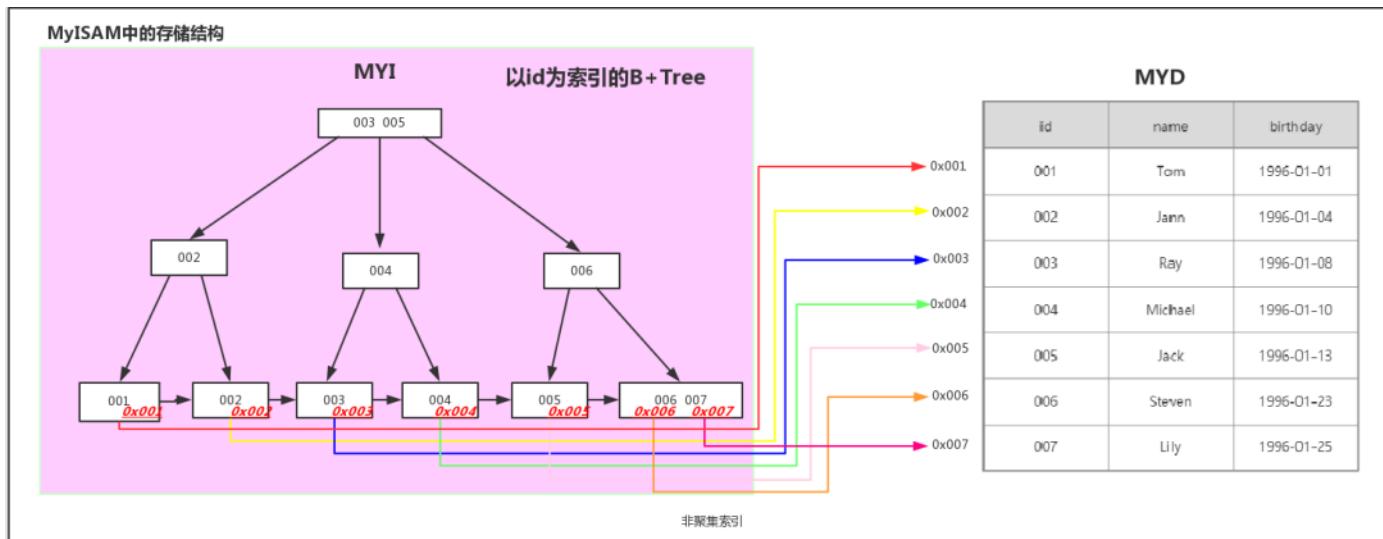


结合存储引擎来讨论（一般默认使用B+Tree）

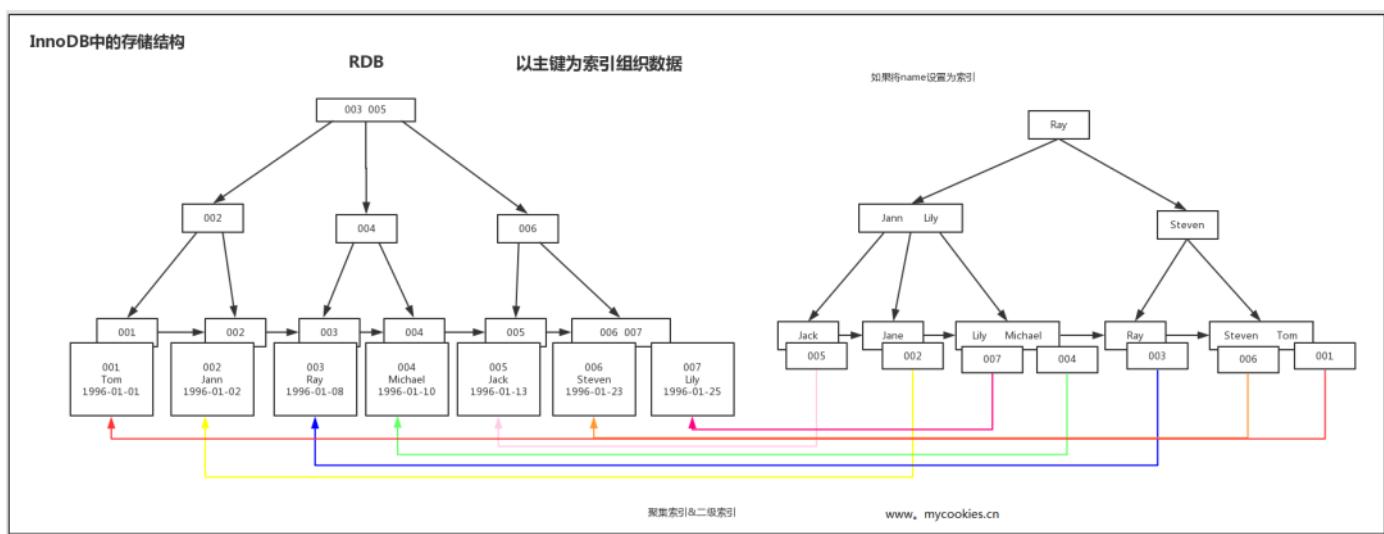
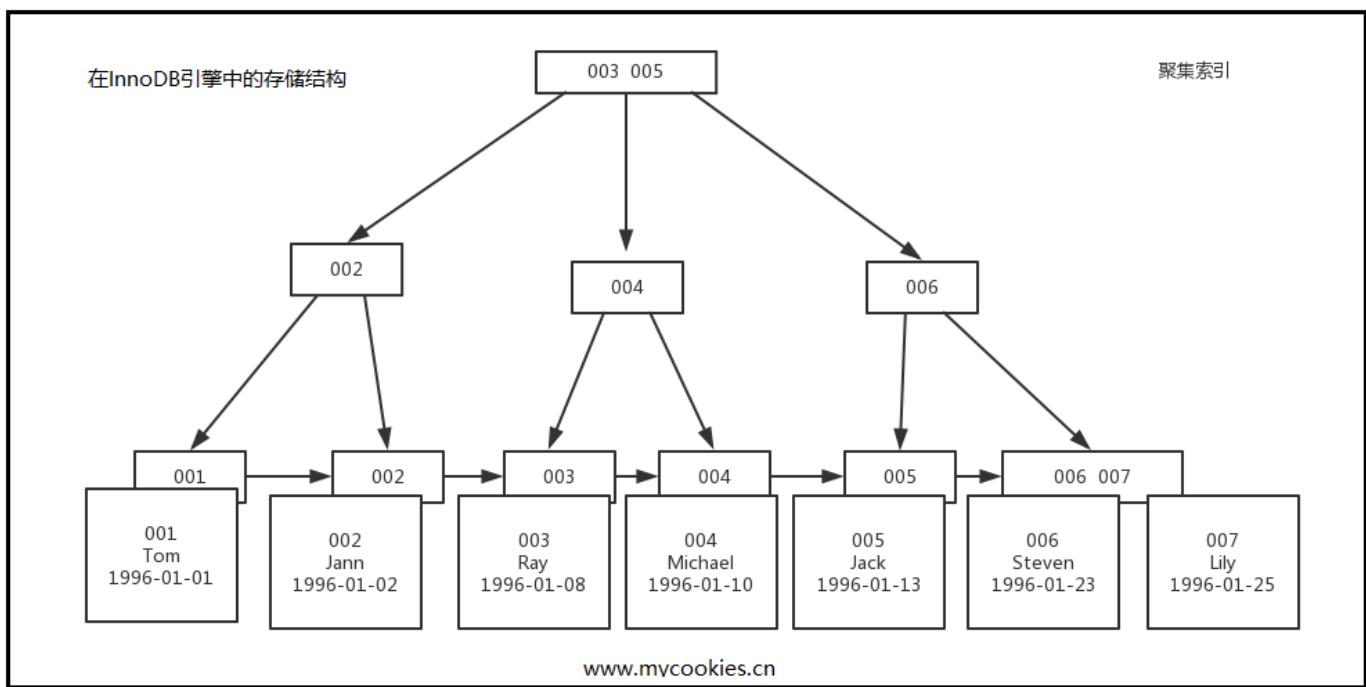
案例：假设有一张学生表，id为主键

id	name	birthday
1	Tom	1996-01-01
2	Jann	1996-01-04
3	Ray	1996-01-08
4	Michael	1996-01-10
5	Jack	1996-01-13
6	Steven	1996-01-23
7	Lily	1996-01-25

在MyISAM引擎中的实现（二级索引也是这样实现的）



在InnoDB中的实现



三、问题

问：为什么索引结构默认使用B-Tree，而不是hash，二叉树，红黑树？

hash：虽然可以快速定位，但是没有顺序，IO复杂度高。

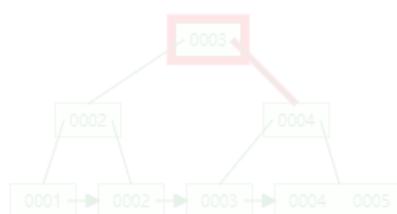
二叉树：树的高度不均匀，不能自平衡，查找效率跟数据有关（树的高度），并且IO代价高。

红黑树：树的高度随着数据量增加而增加，IO代价高。

问：为什么官方建议使用自增长主键作为索引。

结合B+Tree的特点，自增主键是连续的，在插入过程中尽量减少页分裂，即使要进行页分裂，也只会分裂很少一部分。并且能减少数据的移动，每次插入都是插入到最后。总之就是减少分裂和移动的频率。

插入连续的数据：



插入非连续的数据

tom

-END-

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

1. [Java后端优质文章整理](#)
2. [JDK 13 新特性一览](#)
3. [14 个实用的数据库设计技巧](#)
4. [面试官：Redis 内存满了怎么办？](#)
5. [如何设计 API 接口，实现统一格式返回？](#)



Java后端

长按识别二维码，关注我的公众号

[喜欢文章，点个在看](#)

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

建议收藏！写给程序员的 MySQL 面试高频 100 问

Java后端 2019-12-09

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

来源 | juejin.im/post/5d351303f265da1bd30596f9

前言

本文主要受众为开发人员,所以不涉及到MySQL的服务部署等操作,且内容较多,大家准备好耐心和瓜子矿泉水.

前一阵系统的学习了一下MySQL,也有一些实际操作经验,偶然看到一篇和MySQL相关的面试文章,发现其中的一些问题自己也回答不好,虽然知识点大部分都知道,但是无法将知识串联起来.

因此决定搞一个MySQL灵魂100问,试着用回答问题的方式,让自己对知识点的理解更加深入一点.

此文不会事无巨细的从select的用法开始讲解mysql,主要针对的是开发人员需要知道的一些MySQL的知识点

主要包括索引,事务,优化等方面,以在面试中高频的问句形式给出答案.

索引相关

关于MySQL的索引,曾经进行过一次总结,文章链接在这里 [Mysql索引原理及其优化](#).

1. 什么是索引?

索引是一种数据结构,可以帮助我们快速的进行数据的查找.

2. 索引是个什么样的数据结构呢?

索引的数据结构和具体存储引擎的实现有关, 在MySQL中使用较多的索引有Hash索引,B+树索引等,而我们经常使用的InnoDB存储引擎的默认索引实现为:B+树索引.

3. Hash索引和B+树所有有什么区别或者说优劣呢?

首先要知道Hash索引和B+树索引的底层实现原理:

hash索引底层就是hash表,进行查找时,调用一次hash函数就可以获取到相应的键值,之后进行回表查询获得实际数据.B+树底层实现是多路平衡查找树.

对于每一次的查询都是从根节点出发,查找到叶子节点方可以获得所查键值,然后根据查询判断是否需要回表查询数据.

那么可以看出他们有以下的不同:

- hash索引进行等值查询更快(一般情况下),但是却无法进行范围查询.

因为在hash索引中经过hash函数建立索引之后,索引的顺序与原顺序无法保持一致,不能支持范围查询.

而B+树的所有节点都遵循(左节点小于父节点,右节点大于父节点,多叉树也类似),天然支持范围.

- hash索引不支持使用索引进行排序,原理同上.
- hash索引不支持模糊查询以及多列索引的最左前缀匹配.原理也是因为hash函数的不可预测,AAA和AAAB的索引没有相关性.
- hash索引任何时候都避免不了回表查询数据,而B+树在符合某些条件(聚簇索引,覆盖索引等)的时候可以只通过索引完成查询.
- hash索引虽然在等值查询上较快,但是不稳定.性能不可预测,当某个键值存在大量重复的时候,发生hash碰撞,此时效率可能极差.而B+树的查询效率比较稳定,对于所有的查询都是从根节点到叶子节点,且树的高度较低.

因此,在大多数情况下,直接选择B+树索引可以获得稳定且较好的查询速度.而不需要使用hash索引.

4. 上面提到了B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据,什么是聚簇索引?

在B+树的索引中,叶子节点可能存储了当前的key值,也可能存储了当前的key值以及整行的数据,这就是聚簇索引和非聚簇索引.

在InnoDB中,只有主键索引是聚簇索引,如果没有主键,则挑选一个唯一键建立聚簇索引.如果没有唯一键,则隐式的生成一个键来建立聚簇索引.

当查询使用聚簇索引时,在对应的叶子节点,可以获取到整行数据,因此不用再次进行回表查询.

5. 非聚簇索引一定会回表查询吗?

不一定,这涉及到查询语句所要求的字段是否全部命中了索引,如果全部命中了索引,那么就不必再进行回表查询.

举个简单的例子,假设我们在员工表的年龄上建立了索引,那么当进行 `select age from employee where age < 20` 的查询时,在索引的叶子节点上,已经包含了age信息,不会再次进行回表查询.

6. 在建立索引的时候,都有哪些需要考虑的因素呢?

建立索引的时候一般要考虑到字段的使用频率,经常作为条件进行查询的字段比较适合.如果需要建立联合索引的话,还需要考虑联合索引中的顺序.

此外也要考虑其他方面,比如防止过多的所有对表造成太大的压力.这些都和实际的表结构以及查询方式有关.

7. 联合索引是什么?为什么需要注意联合索引中的顺序?

MySQL可以使用多个字段同时建立一个索引,叫做联合索引.在联合索引中,如果想要命中索引,需要按照建立索引时的字段顺序挨个使用,否则无法命中索引.

具体原因为:

MySQL使用索引时需要索引有序,假设现在建立了"name,age,school"的联合索引

那么索引的排序为: 先按照name排序,如果name相同,则按照age排序,如果age的值也相等,则按照school进行排序.

当进行查询时,此时索引仅仅按照name严格有序,因此必须首先使用name字段进行等值查询,之后对于匹配到的列而言,其按照age字段严格有序,此时可以使用age字段用做索引查找,以此类推.

因此在建立联合索引的时候应该注意索引列的顺序,一般情况下,将查询需求频繁或者字段选择性高的列放在前面.此外可以根据特例的查询或者表结构进行单独的调整.

8. 创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?

MySQL提供了explain命令来查看语句的执行计划,MySQL在执行某个语句之前,会将该语句过一遍查询优化器,之后会拿到对语句的分析,也就是执行计划,其中包含了许多信息.

可以通过其中和索引有关的信息来分析是否命中了索引,例如possible_key,key,key_len等字段,分别说明了此语句可能会使用的索引,实际使用的索引以及使用的索引长度.

9. 那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?

- 使用不等于查询
- 列参与了数学运算或者函数
- 在字符串like时左边是通配符.类似于'%aaa'.
- 当mysql分析全表扫描比使用索引快的时候不使用索引.
- 当使用联合索引,前面一个条件为范围查询,后面的即使符合最左前缀原则,也无法使用索引.

以上情况,MySQL无法使用索引.

事务相关

1. 什么是事务?

理解什么是事务最经典的就是转账的栗子,相信大家也都了解,这里就不再说一边了.

事务是一系列的操作,他们要符合ACID特性.最常见的理解就是:事务中的操作要么全部成功,要么全部失败.但是只是这样还不够的.

2. ACID是什么?可以详细说一下吗?

A=Atomicity

原子性,就是上面说的,要么全部成功,要么全部失败.不可能只执行一部分操作.

C=Consistency

系统(数据库)总是从一个一致性的状态转移到另一个一致性的状态,不会存在中间状态.

I=Isolation

隔离性: 通常来说:一个事务在完全提交之前,对其他事务是不可见的.注意前面的通常来说加了红色,意味着有例外情况.

D=Durability

持久性,一旦事务提交,那么就永远是这样子了,哪怕系统崩溃也不会影响到这个事务的结果.

3. 同时有多个事务在进行会怎么样呢?

多事务的并发进行一般会造成以下几个问题:

- 脏读: A事务读取到了B事务未提交的内容,而B事务后面进行了回滚.
- 不可重复读: 当设置A事务只能读取B事务已经提交的部分,会造成在A事务内的两次查询,结果竟然不一样,因为在此期间B事务进行了提交操作.
- 幻读: A事务读取了一个范围的内容,而同时B事务在此期间插入了一条数据.造成"幻觉".

4. 怎么解决这些问题呢?MySQL的事务隔离级别了解吗?

MySQL的四种隔离级别如下:

- 未提交读(READ UNCOMMITTED)

这就是上面所说的例外情况了,这个隔离级别下,其他事务可以看到本事务没有提交的部分修改.因此会造成脏读的问题(读取到了其他事务未提交的部分,而之后该事务进行了回滚).

这个级别的性能没有足够大的优势,但是又有很多的问题,因此很少使用.

- 已提交读(READ COMMITTED)

其他事务只能读取到本事务已经提交的部分.这个隔离级别有 不可重复读的问题,在同一个事务内的两次读取,拿到的结果竟然不一样,因为另外一个事务对数据进行了修改.

- REPEATABLE READ(可重复读)

可重复读隔离级别解决了上面不可重复读的问题(看名字也知道),但是仍然有一个新问题,就是幻读

当你读取id> 10 的数据行时,对涉及到的所有行加上了读锁,此时例外一个事务新插入了一条id=11的数据,因为是新插入的,所以不会触发上面的锁的排斥

那么进行本事务进行下一次的查询时会发现有一条id=11的数据,而上次的查询操作并没有获取到,再进行插入就会有主键冲突的问

题.

- SERIALIZABLE(可串行化)

这是最高的隔离级别,可以解决上面提到的所有问题,因为他强制将所有的操作串行执行,这会导致并发性能极速下降,因此也不是很常用.

5. Innodb使用的是哪种隔离级别呢?

InnoDB默认使用的是可重复读隔离级别.

6. 对MySQL的锁了解吗?

当数据库有并发事务的时候,可能会产生数据的不一致,这时候需要一些机制来保证访问的次序,锁机制就是这样的一个机制.

就像酒店的房间,如果大家随意进出,就会出现多人抢夺同一个房间的情况,而在房间上装上锁,申请到钥匙的人才可以入住并且将房间锁起来,其他人只有等他使用完毕才可以再次使用.

7. MySQL都有哪些锁呢?像上面那样子进行锁定岂不是有点阻碍并发效率了?

从锁的类别上来讲,有共享锁和排他锁.

共享锁: 又叫做读锁. 当用户要进行数据的读取时,对数据加上共享锁.共享锁可以同时加上多个.

排他锁: 又叫做写锁. 当用户要进行数据的写入时,对数据加上排他锁.排他锁只可以加一个,他和其他的排他锁,共享锁都相斥.

用上面的例子来说就是用户的行为有两种,一种是来看房,多个用户一起看房是可以接受的. 一种是真正的入住一晚,在这期间,无论是想入住的还是想看房的都不可以.

锁的粒度取决于具体的存储引擎,InnoDB实现了行级锁,页级锁,表级锁.

他们的加锁开销从大大小,并发能力也是从大到小.

表结构设计

1. 为什么要尽量设定一个主键?

主键是数据库确保数据行在整张表唯一性的保障,即使业务上本张表没有主键,也建议添加一个自增长的ID列作为主键.

设定了主键之后,在后续的删改查的时候可能更加快速以及确保操作数据范围安全.

2. 主键使用自增ID还是UUID?

推荐使用自增ID,不要使用UUID.

因为在InnoDB存储引擎中,主键索引是作为聚簇索引存在的

也就是说,主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序)

如果主键索引是自增ID,那么只需要不断向后排列即可,如果是UUID,由于到来的ID与原来的大小不确定,会造成非常多的数据插入,数据移动,然后导致产生很多的内存碎片,进而造成插入性能的下降.

总之,在数据量大一些的情况下,用自增主键性能会好一些.

图片来源于《高性能MySQL》:其中默认后缀为使用自增ID,_uuid为使用UUID为主键的测试,测试了插入100w行和300w行的性能.

表 5-1: 向InnoDB表插入数据的测试结果

表名	行数	时间 (秒)	索引大小 (MB)
userinfo	1 000 000	137	342
userinfo_uuid	1 000 000	180	544
userinfo	3 000 000	1233	1036
userinfo_uuid	3 000 000	4525	1707

关于主键是聚簇索引,如果没有主键,InnoDB会选择一个唯一键来作为聚簇索引,如果没有唯一键,会生成一个隐式的主键.

If you define a PRIMARY KEY on your table, InnoDB uses it as the clustered index.

If you do not define a PRIMARY KEY for your table, MySQL picks the first UNIQUE index that has only NOT NULL columns as the primary key and InnoDB uses it as the clustered index.

3. 字段为什么要求定义为not null?

MySQL官网这样介绍:

NULL columns require additional space in the row to record whether their values are NULL. For MyISAM tables, each NULL column takes one bit extra, rounded up to the nearest byte.

null值会占用更多的字节,且会在程序中造成很多与预期不符的情况.

4. 如果要存储用户的密码散列,应该使用什么字段进行存储?

密码散列,盐,用户身份证号等固定长度的字符串应该使用char而不是varchar来存储,这样可以节省空间且提高检索效率.

存储引擎相关

1. MySQL支持哪些存储引擎?

MySQL支持多种存储引擎,比如InnoDB,MyISAM,Memory,Archive等等.

在大多数的情况下,直接选择使用InnoDB引擎都是最合适的,InnoDB也是MySQL的默认存储引擎.

1. InnoDB和MyISAM有什么区别?

- InnoDB支持事物, 而MyISAM不支持事物
- InnoDB支持行级锁, 而MyISAM支持表级锁
- InnoDB支持MVCC, 而MyISAM不支持
- InnoDB支持外键, 而MyISAM不支持
- InnoDB不支持全文索引, 而MyISAM支持。

零散问题

1. MySQL中的varchar和char有什么区别.

char是一个定长字段,假如申请了 **char(10)** 的空间,那么无论实际存储多少内容.该字段都占用10个字符,而varchar是变长的

也就是说申请的只是最大长度,占用的空间为实际字符长度+1,最后一个字符存储使用了多长的空间.

在检索效率上来讲,char > varchar,因此在使用中,如果确定某个字段的值的长度,可以使用char,否则应该尽量使用varchar.例如存储用户MD5加密后的密码,则应该使用char.

2. varchar(10)和int(10)代表什么含义?

varchar的10代表了申请的空间长度,也是可以存储的数据的最大长度,而int的10只是代表了展示的长度,不足10位以0填充.

也就是说,int(1)和int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示.

3. MySQL的binlog有几种录入格式?分别有什么区别?

有三种格式,statement,row和mixed.

- statement模式下,记录单元为语句.即每一个sql造成的影响会记录.由于sql的执行是有上下文的,因此在保存的时候需要保存相关的信息,同时还有一些使用了函数之类的语句无法被记录复制.
- row级别下,记录单元为每一行的改动,基本是可以全部记下来但是由于很多操作,会导致大量行的改动(比如alter table),因此这种模式的文件保存的信息太多,日志量太大.
- mixed. 一种折中的方案,普通操作使用statement记录,当无法使用statement的时候使用row.

此外,新版的MySQL中对row级别也做了一些优化,当表结构发生变化的时候,会记录语句而不是逐行记录.

4. 超大分页怎么处理?

超大的分页一般从两个方向上来解决.

- 数据库层面,这也是我们主要集中关注的(虽然收效没那么大)

类似于 `select * from table where age > 20 limit 1000000,10` 这种查询其实也是有可以优化的余地的.

这条语句需要load1000000数据然后基本上全部丢弃,只取10条当然比较慢.

我们可以修改为 `select * from table where id in (select id from table where age > 20 limit 1000000,10)`

这样虽然也load了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快.

同时如果ID连续的好,我们还可以 `select * from table where id > 1000000 limit 10`,效率也是不错的

优化的可能性有许多种,但是核心思想都一样,就是减少load的数据.

- 从需求的角度减少这种请求……主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止ID泄漏且连续被人恶意攻击.

解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至redis等k-V数据库中,直接返回即可.

在阿里巴巴《Java开发手册》中,对超大分页的解决办法是类似于上面提到的第一种.

7. 【推荐】利用延迟关联或者子查询优化超多分页场景。

说明 : MySQL 并不是跳过 offset 行 ,而是取 offset+N 行 ,然后返回放弃前 offset 行 ,返回 N 行 ,那当 offset 特别大的时候 ,效率就非常的低下 ,要么控制返回的总页数 ,要么对超过特定阈值的页数进行 SQL 改写。

正例 : 先快速定位需要获取的 id 段 ,然后再关联 :

```
SELECT a.* FROM 表 1 a, (select id from 表 1 where 条件 LIMIT 100000,20 ) b where a.id=b.id
```

5. 关心过业务系统里面的sql耗时吗?统计过慢查询吗?对慢查询都怎么优化过?

在业务系统中,除了使用主键进行的查询,其他的我都会在测试库上测试其耗时,慢查询的统计主要由运维在做,会定期将业务中的慢查询反馈给我们.

慢查询的优化首先要搞明白慢的原因是什么?是查询条件没有命中索引?是load了不需要的数据列?还是数据量太大?

所以优化也是针对这三个方向来的,

- 首先分析语句,看看是否load了额外的数据,可能是查询了多余的行并且抛弃掉了,可能是加载了许多结果中并不需要的列,对语句进行分析以及重写.

- 分析语句的执行计划,然后获得其使用索引的情况,之后修改语句或者修改索引,使得语句可以尽可能的命中索引.
- 如果对语句的优化已经无法进行,可以考虑表中的数据量是否太大,如果是的话可以进行横向或者纵向的分表.

6. 上面提到横向分表和纵向分表,可以分别举一个适合他们的例子吗?

横向分表是按行分表.假设我们有一张用户表,主键是自增ID且同时是用户的ID.数据量较大,有1亿多条,那么此时放在一张表里的查询效果就不太理想.

我们可以根据主键ID进行分表,无论是按尾号分,或者按ID的区间分都是可以的.

假设按照尾号0-99分为100个表,那么每张表中的数据就仅有100w.这时的查询效率无疑是可以满足要求的.

纵向分表是按列分表.假设我们现在有一张文章表.包含字段 **id-摘要-内容**.而系统中的展示形式是刷新出一个列表,列表中仅包含标题和摘要

当用户点击某篇文章进入详情时才需要正文内容.此时,如果数据量大,将内容这个很大且不经常使用的列放在一起会拖慢原表的查询速度.

我们可以将上面的表分为两张. **id-摘要**, **id-内容**.当用户点击详情,那主键再来取一次内容即可.而增加的存储量只是很小的主键字段.代价很小.

当然,分表其实和业务的关联度很高,在分表之前一定要做好调研以及benchmark.不要按照自己的猜想盲目操作.

7. 什么是存储过程? 有哪些优缺点?

存储过程是一些预编译的SQL语句。

1、更加直白的理解: 存储过程可以说是一个记录集, 它是由一些T-SQL语句组成的代码块

这些T-SQL语句代码像一个方法一样实现一些功能(对单表或多表的增删改查), 然后再给这个代码块取一个名字, 在用到这个功能的时候调用他就行了。

2、存储过程是一个预编译的代码块, 执行效率比较高, 一个存储过程替代大量T_SQL语句, 可以降低网络通信量, 提高通信速率, 可以一定程度上确保数据安全

但是,在互联网项目中,其实是不太推荐存储过程的,比较出名的就是阿里的《Java开发手册》中禁止使用存储过程

我个人的理解是,在互联网项目中,迭代太快,项目的生命周期也比较短,人员流动相比于传统的项目也更加频繁

在这样的情况下,存储过程的管理确实是没有那么方便,同时,复用性也没有写在服务层那么好.

8. 说一说三个范式

第一范式: 每个列都不可以再拆分.

第二范式: 非主键列完全依赖于主键,而不能是依赖于主键的一部分.

第三范式: 非主键列只依赖于主键,不依赖于其他非主键.

在设计数据库结构的时候,要尽量遵守三范式,如果不遵守,必须有足够的理由.比如性能.事实上我们经常会为了性能而妥协数据库的设计.

9. MyBatis 中的

乱入了一个奇怪的问题……我只是想单独记录一下这个问题,因为出现频率太高了.

会将传入的内容当做字符串,而\$会直接将传入值拼接在sql语句中.

所以#可以在一定程度上预防sql注入攻击.

【END】

如果看到这里,说明你喜欢这篇文章,请[转发、点赞](#).微信搜索「web_resource」,关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

[1. 为什么你学不会递归?](#)

[2. MySQL:Left Join 避坑指南](#)

[3. AJAX 请求真的不安全么？](#)

[4. 一个女生不主动联系你还有机会吗？](#)

[5. 团队开发中 Git 最佳实践](#)

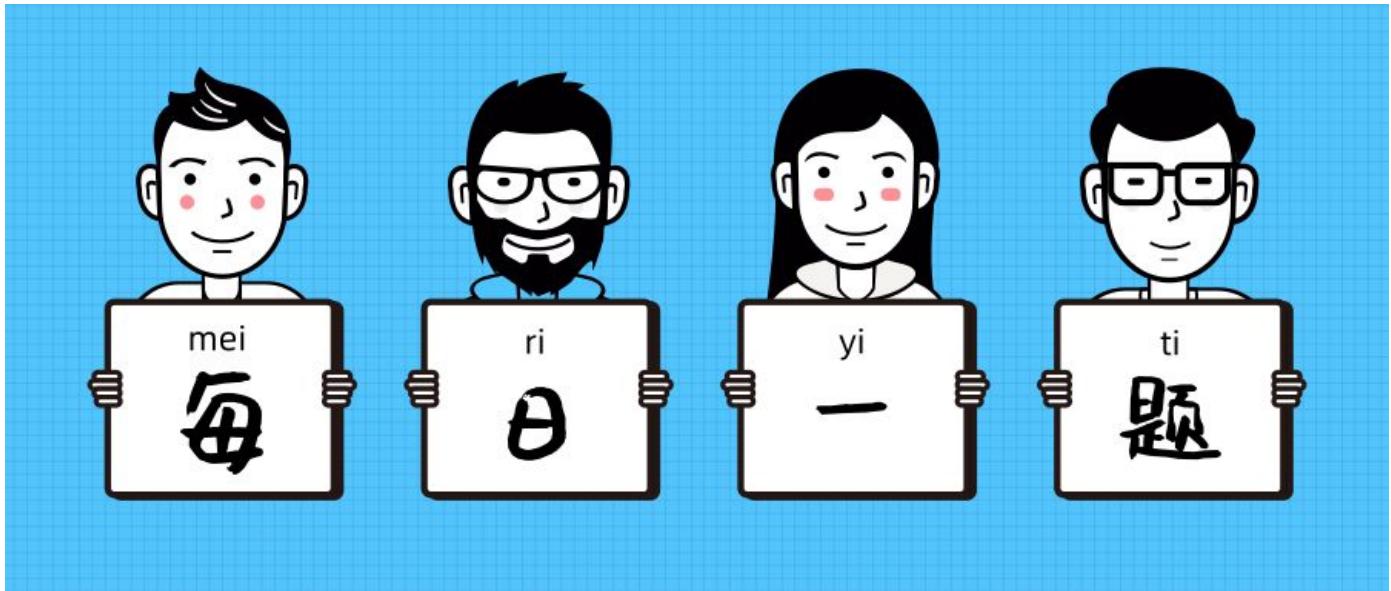


喜欢文章，点个在看 

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

每日一题：你有没有做过 MySQL 读写分离？

Java后端 2019-09-22



原文来自 GitHub 开源社区 Doocs，欢迎 Star 此项目，如果你有独到的见解，同样可以参与贡献此项目。

面试题

你们有没有做 MySQL 读写分离？如何实现 MySQL 的读写分离？MySQL 主从复制原理的是啥？如何解决 MySQL 主从同步的延时问题？

面试官心理分析

高并发这个阶段，肯定是需要做读写分离的，啥意思？因为实际上大部分的互联网公司，一些网站，或者是 app，其实都是读多写少。所以针对这个情况，就是写一个主库，但是主库挂多个从库，然后从多个从库来读，那不就可以支撑更高的读并发压力了吗？

面试题剖析

如何实现 MySQL 的读写分离？

其实很简单，就是基于主从复制架构，简单来说，就搞一个主库，挂多个从库，然后我们就单单只是写主库，然后主库会自动把数据给同步到从库上去。

MySQL 主从复制原理的是啥？

主库将变更写入 binlog 日志，然后从库连接到主库之后，从库有一个 IO 线程，将主库的 binlog 日志拷贝到自己本地，写入一个 relay 中继日志中。接着从库中有一个 SQL 线程会从中继日志读取 binlog，然后执行 binlog 日志中的内容，也就是在自己本地再次执行一遍 SQL，这样就可以保证自己跟主库的数据是一样的。

这里有一个非常重要的一点，就是从库同步主库数据的过程是串行化的，也就是说主库上并行的操作，在从库上会串行执行。所以这就是一个非常重要的点了，由于从库从主库拷贝日志以及串行执行 SQL 的特点，在高并发场景下，从库的数据一定会比主库慢一些，是有延时的。所以经常出现，刚写入主库的数据可能是读不到的，要过几十毫秒，甚至几百毫秒才能读取到。

而且这里还有另外一个问题，就是如果主库突然宕机，然后恰好数据还没同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。

所以 MySQL 实际上在这一块有两个机制，一个是**半同步复制**，用来解决主库数据丢失问题；一个是**并行复制**，用来解决主从同步延时问题。

这个所谓**半同步复制**，也叫 `semi-sync` 复制，指的就是主库写入 binlog 日志之后，就会将**强制**此时立即将数据同步到从库，从库将日志写入自己本地的 relay log 之后，接着会返回一个 ack 给主库，主库接收到**至少一个从库**的 ack 之后才会认为写操作完成了。

所谓**并行复制**，指的是从库开启多个线程，并行读取 relay log 中不同库的日志，然后**并行重放不同库的日志**，这是库级别的并行。

MySQL 主从同步延时问题（精华）

以前线上确实处理过因为主从同步延时问题而导致的线上的 bug，属于小型的生产事故。

是这个么场景。有个同学是这样写代码逻辑的。先插入一条数据，再把它查出来，然后更新这条数据。在生产环境高峰期，写并发达到了 2000/s，这个时候，主从复制延时大概是在小几十毫秒。线上会发现，每天总有那么一些数据，我们期望更新一些重要的数据状态，但在高峰期时候却没更新。用户跟客服反馈，而客服就会反馈给我们。

我们通过 MySQL 命令：】

```
1 show status
```

查看 `Seconds_Behind_Master`，可以看到从库复制主库的数据落后了几 ms。

一般来说，如果主从延迟较为严重，有以下解决方案：

- 分库，将一个主库拆分为多个主库，每个主库的写并发就减少了几倍，此时主从延迟可以忽略不计。
- 打开 MySQL 支持的并行复制，多个库并行复制。如果说某个库的写入并发就是特别高，单库写并发达到了 2000/s，并行复制还是没意义。
- 重写代码，写代码的同学，要慎重，插入数据时立马查询可能查不到。
- 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询设置直连主库。**不推荐**这种方法，你要是这么搞，读写分离的意义就丧失了。

- END -

如果看到这里，说明你喜欢这篇文章，帮忙[转发](#)一下吧，感谢。微信搜索「web_resource」，关注后即可获取每日一题的推送。

推荐阅读

1. 每日一题：消息队列面试常问题目
2. 每日一题：如何保证消息队列的高可用？
3. 每日一题：如何设计一个高并发系统？
4. 每日一题：为什么要进行系统拆分？
5. 接私活必备的 10 个开源项目



Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看✿

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

面试必问的 MySQL 四种隔离级别，看完吊打面试官

游泳的石头 Java后端 2019-11-30

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | 游泳的石头

来源 | www.jianshu.com/p/8d735db9c2c0

什么是事务

事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤消。也就是事务具有原子性，一个事务中的一系列的操作要么全部成功，要么一个都不做。

事务的结束有两种，当事务中的所有步骤全部成功执行时，事务提交。如果其中一个步骤失败，将发生回滚操作，撤消撤消之前到事务开始时的所有操作。

事务的 ACID

事务具有四个特征：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。这四个特性简称为 ACID 特性。

- 原子性。事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做
- 一致性。事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是不一致的状态。
- 隔离性。一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性。也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

Mysql的四种隔离级别

SQL标准定义了4类隔离级别，包括了一些具体规则，用来限定事务内外的哪些改变是可见的，哪些是不可见的。低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。

Read Uncommitted (读取未提交内容)

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

Read Committed (读取提交内容)

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能有新的commit，所以同一select可能返回不同结果。

Repeatable Read (可重读)

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影”行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。

Serializable (可串行化)

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

这四种隔离级别采取不同的锁类型来实现，若读取的是同一个数据的话，就容易发生问题。例如：

- 脏读(Dirty Read):某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。
- 不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中插入了一个事务更新的原有的数据。
- 幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就有几列数据是未查询出来的，如果此时插入和另外一个事务插入的数据，就会报错。

在MySQL中，实现了这四种隔离级别，分别有可能产生问题如下所示：

隔离级别	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	✗	√	√
Repeatable read	✗	✗	√
Serializable	✗	✗	✗

测试Mysql的隔离级别

下面，将利用MySQL的客户端程序，我们分别来测试一下这几种隔离级别。

测试数据库为demo，表为test；表结构：

```
CREATE TABLE `test` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `num` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
```

两个命令行客户端分别为A，B；不断改变A的隔离级别，在B端修改数据。

将A的隔离级别设置为read uncommitted(未提交读)

```
mysql> set session transaction isolation level read uncommitted;
Query OK, 0 rows affected (0.00 sec)
```

A：启动事务，此时数据为初始状态

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| READ-UNCOMMITTED |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B: 启动事务，更新数据，但不提交

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update test set num=10 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

A: 再次读取数据，发现数据已经被修改了，这就是所谓的“脏读”

```
mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B: 回滚事务

```
mysql> rollback;
Query OK, 0 rows affected (0.00 sec)
```

A: 再次读数据，发现数据变回初始状态

```
mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

经过上面的实验可以得出结论，事务B更新了一条记录，但是没有提交，此时事务A可以查询出未提交记录。造成脏读现象。未提交读是最低的隔离级别。

将客户端A的事务隔离级别设置为read committed(已提交读)

```
mysql> set session transaction isolation level read committed;
Query OK, 0 rows affected (0.00 sec)
```

A: 启动事务，此时数据为初始状态

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| READ-COMMITTED |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B: 启动事务，更新数据，但不提交

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update test set num=10 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

A: 再次读数据，发现数据未被修改

```
mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B: 提交事务

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

A: 再次读取数据，发现数据已发生变化，说明B提交的修改被事务中的A读到了，这就是所谓的“不可重复读”

```
mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

经过上面的实验可以得出结论，已提交读隔离级别解决了脏读的问题，但是出现了不可重复读的问题，即事务A在两次查询的数据不一致，因为在两次查询之间事务B更新了一条数据。已提交读只允许读取已提交的记录，但不要求可重复读。

将A的隔离级别设置为repeatable read(可重复读)

```
mysql> set session transaction isolation level repeatable read;
Query OK, 0 rows affected (0.00 sec)
```

A: 启动事务，此时数据为初始状态

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B: 启动事务，更新数据，但不提交

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> update test set num=10 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 10 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

A: 再次读取数据，发现数据未被修改

```
mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B: 提交事务

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

A: 再次读取数据，发现数据依然未发生变化，这说明这次可以重复读了

```
mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B：插入一条新的数据，并提交

```
mysql> insert into test (num) value(4);
Query OK, 1 row affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
+---+---+
4 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

A：再次读取数据，发现数据依然未发生变化，虽然可以重复读了，但是却发现读的不是最新数据，这就是所谓的“幻读”

```
mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

A：提交本次事务，再次读取数据，发现读取正常了

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
+---+---+
4 rows in set (0.00 sec)
```

由以上的实验可以得出结论，可重复读隔离级别只允许读取已提交记录，而且在一个事务两次读取一个记录期间，其他事务部的更新该记录。但该事务不要求与其他事务可串行化。例如，当一个事务可以找到由一个已提交事务更新的记录，但是可能产生幻读问题(注意是可能，因为数据库对隔离级别的实现有所差别)。像以上的实验，就没有出现数据幻读的问题。

将A的隔离级别设置为可串行化(Serializable)

```
mysql> set session transaction isolation level serializable;
Query OK, 0 rows affected (0.00 sec)
```

A：启动事务，此时数据为初始状态

```
mysql> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| SERIALIZABLE   |
+-----+
1 row in set (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test;
+---+---+
| id | num |
+---+---+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+---+---+
3 rows in set (0.00 sec)
```

B：发现B此时进入了等待状态，原因是A的事务尚未提交，只能等待（此时，B可能会发生等待超时）

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into test (num) value(4);
|
```

A：提交事务

```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

B：发现插入成功

```
mysql> insert into test (num) value(4);
Query OK, 1 row affected (3.28 sec)
```

Serializable完全锁定字段，若一个事务来查询同一份数据就必须等待，直到前一个事务完成并解除锁定为止。是完整的隔离级别，会锁定对应的数据表格，因而会有效率的问题。

【END】

如果看到这里，说明你喜欢这篇文章，请[转发、点赞](#)。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



1. 如果我是面试官,我会问你 Spring 那些问题?

2. Spring Boot 整合 Spring-cache

3. 我们再来聊一聊 Java 的单例吧

4. 我采访了一位 Pornhub 工程师

5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

项目中常用到的 19 条 MySQL 优化

zhangqh Java后端 2019-11-19

点击上方 Java后端, 选择 [设为星标](#)

优质文章, 及时送达

作者 | zhangqh

链接 | segmentfault.com/a/1190000012155267

关于MySQL优化方法，网上有不少资料和方法，但是不少质量参差不齐，有些总结的不够到位，内容冗杂。偶尔看到SF，发现了这篇文章，总结得很经典，希望对大家今后开发中有帮助。今天的文章共提到19条常用的MySQL优化方法。

1、EXPLAIN

做MySQL优化，我们要善用EXPLAIN查看SQL执行计划。

下面来个简单的示例，标注（1、2、3、4、5）我们要重点关注的数据：

mysql> explain select create_time,type.secret from t where create_time>=UNIX_TIMESTAMP('2017-11-08');								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	tbl_flight_item	NULL	range	idx_create_time	idx_create_time	5	NULL
								1

- **type列**, 连接类型。一个好的SQL语句至少要达到range级别。杜绝出现all级别。
- **key列**, 使用到的索引名。如果没有选择索引，值是NULL。可以采取强制索引方式。
- **key_len列**, 索引长度。
- **rows列**, 扫描行数。该值是个预估值。
- **extra列**, 详细说明。注意，常见的不太友好的值，如下：Using filesort, Using temporary。

2、SQL语句中IN包含的值不应过多

MySQL对于IN做了相应的优化，即将IN中的常量全部存储在一个数组里面，而且这个数组是排好序的。但是如果数值较多，产生的消耗也是比较大的。再例如：select id from t where num in(1,2,3) 对于连续的数值，能用between就不要用in了；再或者使用连接来替换。

3、SELECT语句务必指明字段名称

SELECT*增加很多不必要的消耗（CPU、IO、内存、网络带宽）；增加了使用覆盖索引的可能性；当表结构发生改变时，前断也需要更新。所以要求直接在select后面接上字段名。

Tips：可以微信搜索：Java后端，关注后加入咱们自己的交流群。

4、当只需要一条数据的时候，使用limit 1

这是为了使EXPLAIN中type列达到const类型

5、如果排序字段没有用到索引，就尽量少排序

6、如果限制条件中其他字段没有索引，尽量少用or

or两边的字段中，如果有一个不是索引字段，而其他条件也不是索引字段，会造成该查询不走索引的情况。很多时候使用union all或者是union（必要的时候）的方式来代替“or”会得到更好的效果。

7、尽量用union all代替union

union和union all的差异主要是前者需要将结果集合并后再进行唯一性过滤操作，这就会涉及到排序，增加大量的CPU运算，加大资源消耗及延迟。当然，union all的前提条件是两个结果集没有重复数据。

8、不使用ORDER BY RAND()

```
select id from `dynamic` order by rand() limit 1000;
```

上面的SQL语句，可优化为：

```
select id from `dynamic` t1 join (select rand() * (select max(id) from `dynamic`) as nid) t2 on t1.id > t2.nid limit 1000;
```

9、区分in和exists、not in和not exists

```
select * from 表A where id in (select id from 表B)
```

上面SQL语句相当于

```
select * from 表A where exists(select * from 表B where 表B.id=表A.id)
```

区分in和exists主要是造成了驱动顺序的改变（这是性能变化的关键），如果是exists，那么以外层表为驱动表，先被访问，如果是IN，那么先执行子查询。所以IN适合于外表大而内表小的情况；EXISTS适合于外表小而内表大的情况。

关于not in和not exists，推荐使用not exists，不仅仅是效率问题，not in可能存在逻辑问题。如何高效的写出一个替代not exists的SQL语句？

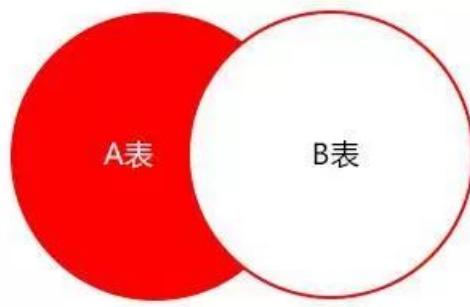
原SQL语句：

```
select colname ... from A表 where a.id not in (select b.id from B表)
```

高效的SQL语句：

```
select colname ... from A表 Left join B表 on where a.id = b.id where b.id is null
```

取出的结果集如下图表示，A表不在B表中的数据：



10、使用合理的分页方式以提高分页的效率

```
select id,name from product limit 866613, 20
```

使用上述SQL语句做分页的时候，可能有人会发现，随着表数据量的增加，直接使用limit分页查询会越来越慢。

优化的方法如下：可以取前一页的最大行数的id，然后根据这个最大的id来限制下一页的起点。比如此例中，上一页最大的id是866612。SQL可以采用如下的写法：

```
select id,name from product where id> 866612 limit 20
```

11、分段查询

在一些用户选择页面中，可能一些用户选择的时间范围过大，造成查询缓慢。主要的原因是扫描行数过多。这个时候可以通过程序，分段进行查询，循环遍历，将结果合并处理进行展示。

如下图这个SQL语句，扫描的行数成百万级以上的时候就可以使用分段查询：

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | dynamic_201702 | range | idx_sch_deptime | idx_sch_deptime | 4 | NULL | 3610145 | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

12、避免在where子句中对字段进行null值判断

对于null的判断会导致引擎放弃使用索引而进行全表扫描。

13、不建议使用%前缀模糊查询

例如LIKE “%name” 或者LIKE “%name%” ，这种查询会导致索引失效而进行全表扫描。但是可以使用LIKE “name%”。

那如何查询%name%？

如下图所示，虽然给secret字段添加了索引，但在explain结果并没有使用：

```

mysql> explain select create_time,type from `tbl_flight_item` where secret like '%9573aa%';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tbl_flight_item | NULL | ALL | NULL | NULL | NULL | NULL | 61439228 | 11.11 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set. 1 warning (0.00 sec)

mysql> show create table tbl_flight_item;
+-----+-----+
| Table | Create Table |
+-----+-----+
| `tbl_flight_item` | CREATE TABLE `tbl_flight_item` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `create_time` int(11) DEFAULT NULL COMMENT '记录时间',
  `type` smallint(6) DEFAULT NULL COMMENT '类型',
  `secret` char(32) DEFAULT NULL COMMENT 'md5',
  `level` char(2) DEFAULT NULL COMMENT '等级',
  PRIMARY KEY (`id`),
  KEY `idx_create_time` (`create_time`),
  KEY `idx_level` (`level`),
  KEY `idx_secret` (`secret`),
  KEY `idx_level` (`level`)
) ENGINE=InnoDB AUTO_INCREMENT=65872985 DEFAULT CHARSET=latin1 |
+-----+-----+

```

那么如何解决这个问题呢，答案：使用全文索引。

在我们查询中经常会用到select id,fnum,fdst from dynamic_201606 where user_name like '%zhangsan%'。这样的语句，普通索引是无法满足查询需求的。庆幸的是在MySQL中，有全文索引来帮助我们。

创建全文索引的SQL语法是：

```
ALTER TABLE `dynamic_201606` ADD FULLTEXT INDEX `idx_user_name`(`user_name`);
```

使用全文索引的SQL语句是：

```
select id,fnum,fdst from dynamic_201606 where match(user_name) against('zhangsan' in boolean mode);
```

注意：在需要创建全文索引之前，请联系DBA确定能否创建。同时需要注意的是查询语句的写法与普通索引的区别。

14、避免在where子句中对字段进行表达式操作

比如：

```
select user_id,user_project from user_base where age*2=36;
```

对字段就行了算术运算，这会造成引擎放弃使用索引，建议改成：

```
select user_id,user_project from user_base where age=36/2;
```

15、避免隐式类型转换

where子句中出现column字段的类型和传入的参数类型不一致的时候发生的类型转换，建议先确定where中的参数类型。

```

mysql> explain select id,create_time from flight_anum where type=43;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | flight_anum | NULL | ALL | idx_type | NULL | NULL | NULL | 2 | 50.00 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 3 warnings (0.00 sec)

mysql> show create table flight_anum;
+-----+-----+
| Table | Create Table |
+-----+-----+
| flight_anum | CREATE TABLE `flight_anum` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`create_time` int(11) DEFAULT NULL COMMENT '记录时间',
`type` varchar(6) DEFAULT NULL COMMENT '类型',
PRIMARY KEY (`id`),
KEY `idx_type` (`type`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=latin1 |
+-----+-----+
1 row in set (0.00 sec)

mysql> explain select id,create_time from flight_anum where type='43';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | flight_anum | NULL | ref | idx_type | idx_type | 9 | const | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

```

16、对于联合索引来说，要遵守最左前缀法则

举例来说索引含有字段id、name、school，可以直接用id字段，也可以id、name这样的顺序，但是name;school都无法使用这个索引。所以在创建联合索引的时候一定要注意索引字段顺序，常用的查询字段放在最前面。

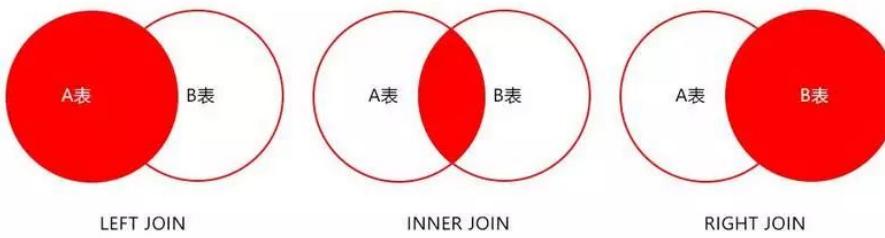
17、必要时可以使用force index来强制查询走某个索引

有的时候MySQL优化器采取它认为合适的索引来检索SQL语句，但是可能它所采用的索引并不是我们想要的。这时就可以采用forceindex来强制优化器使用我们制定的索引。

18、注意范围查询语句

对于联合索引来说，如果存在范围查询，比如between、>、<等条件时，会造成后面的索引字段失效。

19、关于JOIN优化



LEFT JOIN A表为驱动表，INNER JOIN MySQL会自动找出那个数据少的表作用驱动表，RIGHT JOIN B表为驱动表。

注意：

1) MySQL中没有full join，可以用以下方式来解决：

```
select * from A left join B on B.name = A.name where B.name is null union all select * from B;
```

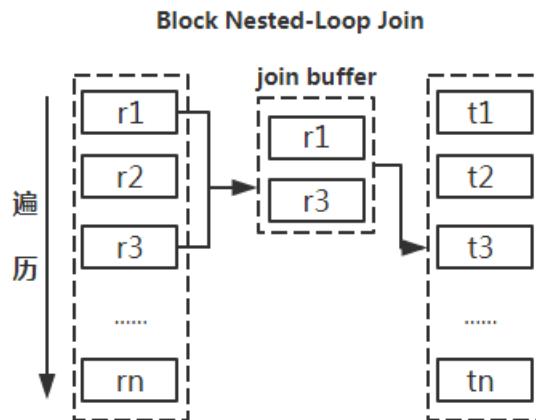
2) 尽量使用inner join，避免left join：

参与联合查询的表至少为2张表，一般都存在大小之分。如果连接方式是inner join，在没有其他过滤条件的情况下MySQL会自动选择小表作为驱动表，但是left join在驱动表的选择上遵循的是左边驱动右边的原则，即left join左边的表名为驱动表。

3) 合理利用索引：

被驱动表的索引字段作为on的限制字段。

4) 利用小表去驱动大表：



从原理图能够直观的看出如果能够减少驱动表的话，减少嵌套循环中的循环次数，以减少 IO总量及CPU运算的次数。

5) 巧用STRAIGHT_JOIN：

inner join是由MySQL选择驱动表，但是有些特殊情况需要选择另个表作为驱动表，比如有group by、order by等「Using filesort」、「Using temporary」时。STRAIGHT_JOIN来强制连接顺序，在STRAIGHT_JOIN左边的表名就是驱动表，右边则是被驱动表。在使用STRAIGHT_JOIN有个前提条件是该查询是内连接，也就是inner join。其他链接不推荐使用STRAIGHT_JOIN，否则可能造成查询结果不准确。

```
SELECT `rl`.*,
       `rl`.`update_time` AS `add_time`,
       `f`.`fnum`,
       `f`.`dst_parking`,
       `f`.`aircraft_num`,
       `f`.`touch_down_runway`,
       `f`.`forg`,
       `f`.`fdst`,
       `f`.`flight_status_code`,
       `fd`.`fdst` AS `fd_fdst`,
       `ds`.`touch_down_runway_source`,
       `ds`.`touch_down_runway_update_time`,
       `u`.`true_name`,
       `a`.`aircraft_model`
FROM _____ AS `rl`
STRAIGHT_JOIN _____ AS `f` ON `rl`.`fid` = `f`.`fid`
LEFT JOIN _____ AS `a` ON `f`.`aircraft_num` = `a`.`aircraft_num`
LEFT JOIN _____ AS `fl` ON `f`.`fid` = `fl`.`arr_fid`
LEFT JOIN _____ AS `fd` ON `fl`.`dep_fid` = `fd`.`fid`
LEFT JOIN _____ AS `ds` ON `f`.`fid` = `ds`.`fid`
LEFT JOIN _____ AS `u` ON `ds`.`touch_down_runway_source` = `u`.`uid`
WHERE `rl`.`op_type` = 0
      AND `f`.`fdst` = 'PVG'
      AND `f`.`fid` != ''
GROUP BY `rl`.`fid`
ORDER BY `rl`.`update_time` DESC
LIMIT 20;
```

这个方式有时能减少3倍的时间。

以上19条MySQL优化方法希望对大家有所帮助！

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！