

01 | 基础架构：一条SQL查询语句是如何执行的？

2018-11-14 林晓斌



你好，我是林晓斌。

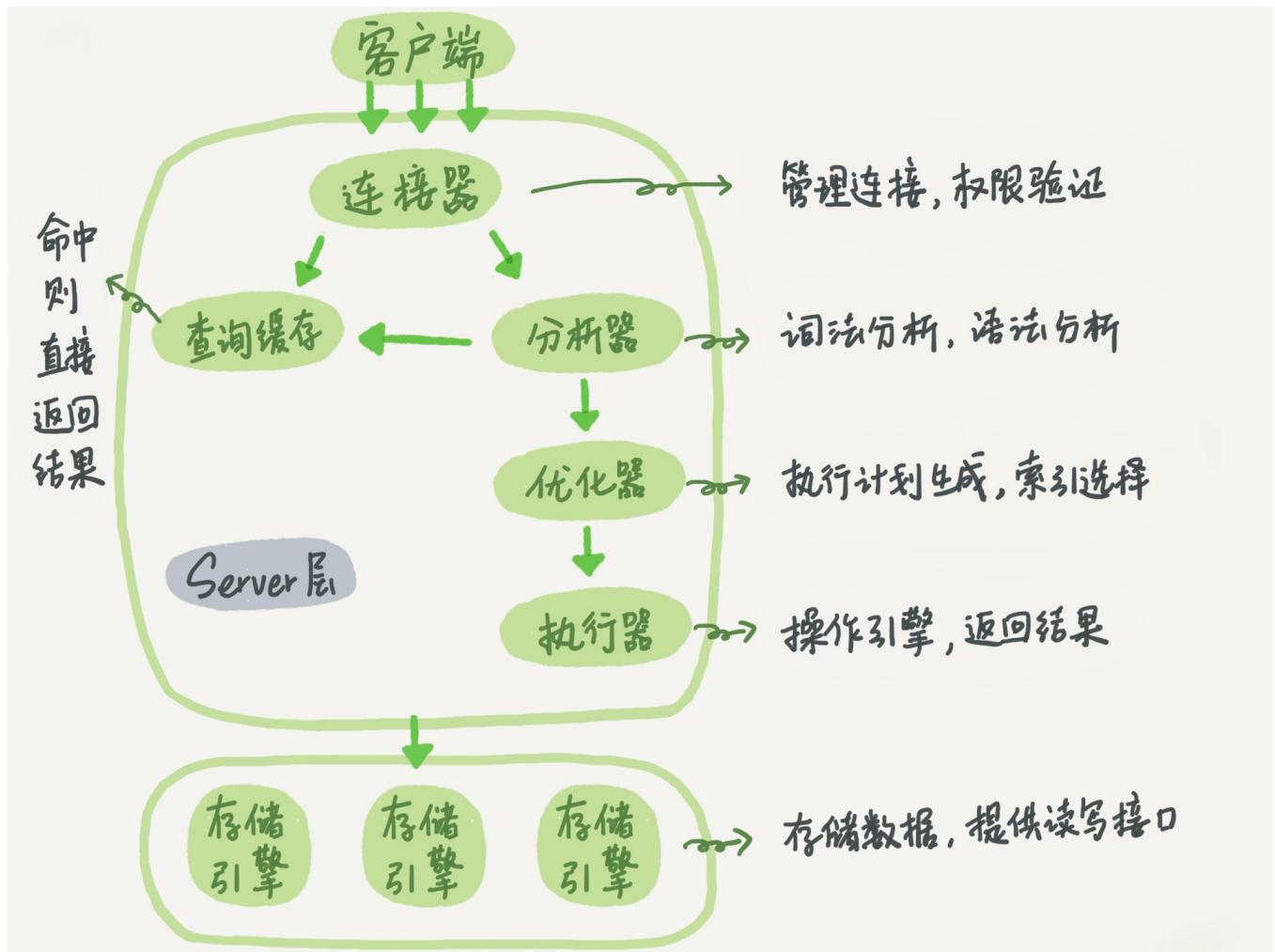
这是专栏的第一篇文章，我想来跟你聊聊MySQL的基础架构。我们经常说，看一个事儿千万不要直接陷入细节里，你应该先鸟瞰其全貌，这样能够帮助你从高维度理解问题。同样，对于MySQL的学习也是这样。平时我们使用数据库，看到的通常都是一个整体。比如，你有个最简单的表，表里只有一个ID字段，在执行下面这个查询语句时：

```
mysql> select * from T where ID=10;
```

我们看到的只是输入一条语句，返回一个结果，却不知道这条语句在MySQL内部的执行过程。

所以今天我想和你一起把MySQL拆解一下，看看里面都有哪些“零件”，希望借由这个拆解过程，让你对MySQL有更深入的理解。这样当我们碰到MySQL的一些异常或者问题时，就能够直戳本质，更为快速地定位并解决问题。

下面我给出的是MySQL的基本架构示意图，从中你可以清楚地看到SQL语句在MySQL的各个功能模块中的执行过程。



MySQL的逻辑架构图

大体来说，MySQL可以分为**Server**层和存储引擎层两部分。

Server层包括连接器、查询缓存、分析器、优化器、执行器等，涵盖MySQL的大多数核心服务功能，以及所有的内置函数（如日期、时间、数学和加密函数等），所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图等。

而存储引擎层负责数据的存储和提取。其架构模式是插件式的，支持InnoDB、MyISAM、Memory等多个存储引擎。现在最常用的存储引擎是InnoDB，它从MySQL 5.5.5版本开始成为了默认存储引擎。

也就是说，你执行`create table`建表的时候，如果不指定引擎类型，默认使用的就是InnoDB。不过，你也可以通过指定存储引擎的类型来选择别的引擎，比如在`create table`语句中使用`engine=memory`，来指定使用内存引擎创建表。不同存储引擎的表数据存取方式不同，支持的功能也不同，在后面的文章中，我们会讨论到引擎的选择。

从图中不难看出，不同的存储引擎共用一个**Server**层，也就是从连接器到执行器的部分。你可以先对每个组件的名字有个印象，接下来我会结合开头提到的那条SQL语句，带你走一遍整个执行流程，依次看下每个组件的作用。

连接器

第一步，你会先连接到这个数据库上，这时候接待你的就是连接器。连接器负责跟客户端建立连接、获取权限、维持和管理连接。连接命令一般是这么写的：

```
mysql -h$ip -P$port -u$user -p
```

输完命令之后，你就需要在交互对话里面输入密码。虽然密码也可以直接跟在-p后面写在命令行中，但这样可能会导致你的密码泄露。如果你连的是生产服务器，强烈建议你不要这么做。

连接命令中的mysql是客户端工具，用来跟服务端建立连接。在完成经典的TCP握手后，连接器就要开始认证你的身份，这个时候用的就是你输入的用户名和密码。

- 如果用户名或密码不对，你就会收到一个"Access denied for user"的错误，然后客户端程序结束执行。
- 如果用户名密码认证通过，连接器会到权限表里面查出你拥有的权限。之后，这个连接里面的权限判断逻辑，都将依赖于此时读到的权限。

这就意味着，一个用户成功建立连接后，即使你用管理员账号对这个用户的权限做了修改，也不会影响已经存在连接的权限。修改完成后，只有再新建的连接才会使用新的权限设置。

连接完成后，如果你没有后续的动作，这个连接就处于空闲状态，你可以在show processlist命令中看到它。文本中这个图是show processlist的结果，其中的Command列显示为“Sleep”的这一行，就表示现在系统里面有一个空闲连接。

```
mysql> show processlist;
+----+----+----+----+----+----+----+----+
| Id | User | Host          | db   | Command | Time | State    | Info
+----+----+----+----+----+----+----+----+
| 5  | root | localhost:27710 | test | Sleep   | 16   | NULL     | 
| 6  | root | localhost:27712 | test | Query   | 0    | starting | show processlist
+----+----+----+----+----+----+----+----+
2 rows in set (0.00 sec)
```

客户端如果太长时间没动静，连接器就会自动将它断开。这个时间是由参数wait_timeout控制的，默认值是8小时。

如果在连接被断开之后，客户端再次发送请求的话，就会收到一个错误提醒： Lost connection to MySQL server during query。这时候如果你要继续，就需要重连，然后再执行请求了。

数据库里面，长连接是指连接成功后，如果客户端持续有请求，则一直使用同一个连接。短连接则是指每次执行完很少的几次查询就断开连接，下次查询再重新建立一个。

建立连接的过程通常是比较复杂的，所以我建议你在使用中要尽量减少建立连接的动作，也就是尽量使用长连接。

但是全部使用长连接后，你可能会发现，有些时候MySQL占用内存涨得特别快，这是因为MySQL在执行过程中临时使用的内存是管理在连接对象里面的。这些资源会在连接断开的时候才释放。所以如果长连接累积下来，可能导致内存占用太大，被系统强行杀掉（OOM），从现象看就是MySQL异常重启了。

怎么解决这个问题呢？你可以考虑以下两种方案。

1. 定期断开长连接。使用一段时间，或者程序里面判断执行过一个占用内存的大查询后，断开连接，之后要查询再重连。
2. 如果你用的是MySQL 5.7或更新版本，可以在每次执行一个比较大的操作后，通过执行`mysql_reset_connection`来重新初始化连接资源。这个过程不需要重连和重新做权限验证，但是会将连接恢复到刚刚创建完时的状态。

查询缓存

连接建立完成后，你就可以执行`select`语句了。执行逻辑就会来到第二步：查询缓存。

MySQL拿到一个查询请求后，会先到查询缓存看看，之前是不是执行过这条语句。之前执行过的语句及其结果可能会以key-value对的形式，被直接缓存在内存中。`key`是查询的语句，`value`是查询的结果。如果你的查询能够直接在这个缓存中找到`key`，那么这个`value`就会被直接返回给客户端。

如果语句不在查询缓存中，就会继续后面的执行阶段。执行完成后，执行结果会被存入查询缓存中。你可以看到，如果查询命中缓存，MySQL不需要执行后面的复杂操作，就可以直接返回结果，这个效率会很高。

但是大多数情况下我会建议你不要使用查询缓存，为什么呢？因为查询缓存往往弊大于利。

查询缓存的失效非常频繁，只要有对一个表的更新，这个表上所有的查询缓存都会被清空。因此很可能你费劲地把结果存起来，还没使用呢，就被一个更新全清空了。对于更新压力大的数据库来说，查询缓存的命中率会非常低。除非你的业务就是有一张静态表，很长时间才会更新一次。比如，一个系统配置表，那这张表上的查询才适合使用查询缓存。

好在MySQL也提供了这种“按需使用”的方式。你可以将参数`query_cache_type`设置成`DEMAND`，这样对于默认的SQL语句都不使用查询缓存。而对于你确定要使用查询缓存的语句，可以用`SQL_CACHE`显式指定，像下面这个语句一样：

```
mysql> select SQL_CACHE * from T where ID=10;
```

需要注意的是，MySQL 8.0版本直接将查询缓存的整块功能删掉了，也就是说8.0开始彻底没有这个功能了。

分析器

如果没有命中查询缓存，就要开始真正执行语句了。首先，MySQL需要知道你要做什么，因此需要对SQL语句做解析。

分析器先会做“词法分析”。你输入的是由多个字符串和空格组成的一条SQL语句，MySQL需要识别出里面的字符串分别是什么，代表什么。

MySQL从你输入的"select"这个关键字识别出来，这是一个查询语句。它也要把字符串“T”识别成“表名T”，把字符串“ID”识别成“列ID”。

做完了这些识别以后，就要做“语法分析”。根据词法分析的结果，语法分析器会根据语法规则，判断你输入的这个SQL语句是否满足MySQL语法。

如果你的语句不对，就会收到“You have an error in your SQL syntax”的错误提醒，比如下面这个语句select少打了开头的字母“s”。

```
mysql> elect * from t where ID=1;

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to

```

一般语法错误会提示第一个出现错误的位置，所以你要关注的是紧接“use near”的内容。

优化器

经过了分析器，MySQL就知道你要做什么了。在开始执行之前，还要先经过优化器的处理。

优化器是在表里面有多个索引的时候，决定使用哪个索引；或者在一个语句有多表关联（join）的时候，决定各个表的连接顺序。比如你执行下面这样的语句，这个语句是执行两个表的join：

```
mysql> select * from t1 join t2 using(ID) where t1.c=10 and t2.d=20;
```

- 既可以先从表t1里面取出c=10的记录的ID值，再根据ID值关联到表t2，再判断t2里面d的值是否等于20。
- 也可以先从表t2里面取出d=20的记录的ID值，再根据ID值关联到t1，再判断t1里面c的值是否等于10。

这两种执行方法的逻辑结果是一样的，但是执行的效率会有不同，而优化器的作用就是决定选择

使用哪一个方案。

优化器阶段完成后，这个语句的执行方案就确定下来了，然后进入执行器阶段。如果你还有一些疑问，比如优化器是怎么选择索引的，有没有可能选择错等等，没关系，我会在后面的文章中单独展开说明优化器的内容。

执行器

MySQL通过分析器知道了你要做什么，通过优化器知道了该怎么做，于是就进入了执行器阶段，开始执行语句。

开始执行的时候，要先判断一下你对这个表T有没有执行查询的权限，如果没有，就会返回没有权限的错误，如下所示。

```
mysql> select * from T where ID=10;  
  
ERROR 1142 (42000): SELECT command denied to user 'b'@'localhost' for table 'T'
```

如果有权限，就打开表继续执行。打开表的时候，执行器就会根据表的引擎定义，去使用这个引擎提供的接口。

比如我们这个例子中的表T中，ID字段没有索引，那么执行器的执行流程是这样的：

1. 调用InnoDB引擎接口取这个表的第一行，判断ID值是不是10，如果不是则跳过，如果是则将这行存在结果集中；
2. 调用引擎接口取“下一行”，重复相同的判断逻辑，直到取到这个表的最后一行。
3. 执行器将上述遍历过程中所有满足条件的行组成的记录集作为结果集返回给客户端。

至此，这个语句就执行完成了。

对于有索引的表，执行的逻辑也差不多。第一次调用的是“取满足条件的第一行”这个接口，之后循环取“满足条件的下一行”这个接口，这些接口都是引擎中已经定义好的。

你会在数据库的慢查询日志中看到一个rows_examined的字段，表示这个语句执行过程中扫描了多少行。这个值就是在执行器每次调用引擎获取数据行的时候累加的。

在有些场景下，执行器调用一次，在引擎内部则扫描了多行，因此**引擎扫描行数跟rows_examined并不是完全相同的**。我们后面会专门有一篇文章来讲存储引擎的内部机制，里面有详细的说明。

小结

今天我给你介绍了MySQL的逻辑架构，希望你对一个SQL语句完整执行流程的各个阶段有了一个初步的印象。由于篇幅的限制，我只是用一个查询的例子将各个环节过了一遍。如果你还对每个环节的展开细节存有疑问，也不用担心，后续在实战章节中我还会再提到它们。

我给你留一个问题吧，如果表T中没有字段k，而你执行了这个语句 `select * from T where k=1`，那肯定是会报“不存在这个列”的错误：“Unknown column ‘k’ in ‘where clause’”。你觉得这个错误是在我们上面提到的哪个阶段报出来的呢？

感谢你的收听，欢迎你给我留言，也欢迎分享给更多的朋友一起阅读。



02 | 日志系统：一条SQL更新语句是如何执行的？

2018-11-16 林晓斌



前面我们系统了解了一个查询语句的执行流程，并介绍了执行过程中涉及的处理模块。相信你还记得，一条查询语句的执行过程一般是经过连接器、分析器、优化器、执行器等功能模块，最后到达存储引擎。

那么，一条更新语句的执行流程又是怎样的呢？

之前你可能经常听DBA同事说，MySQL可以恢复到半个月内任意一秒的状态，惊叹的同时，你是不是心中也会不免会好奇，这是怎样做到的呢？

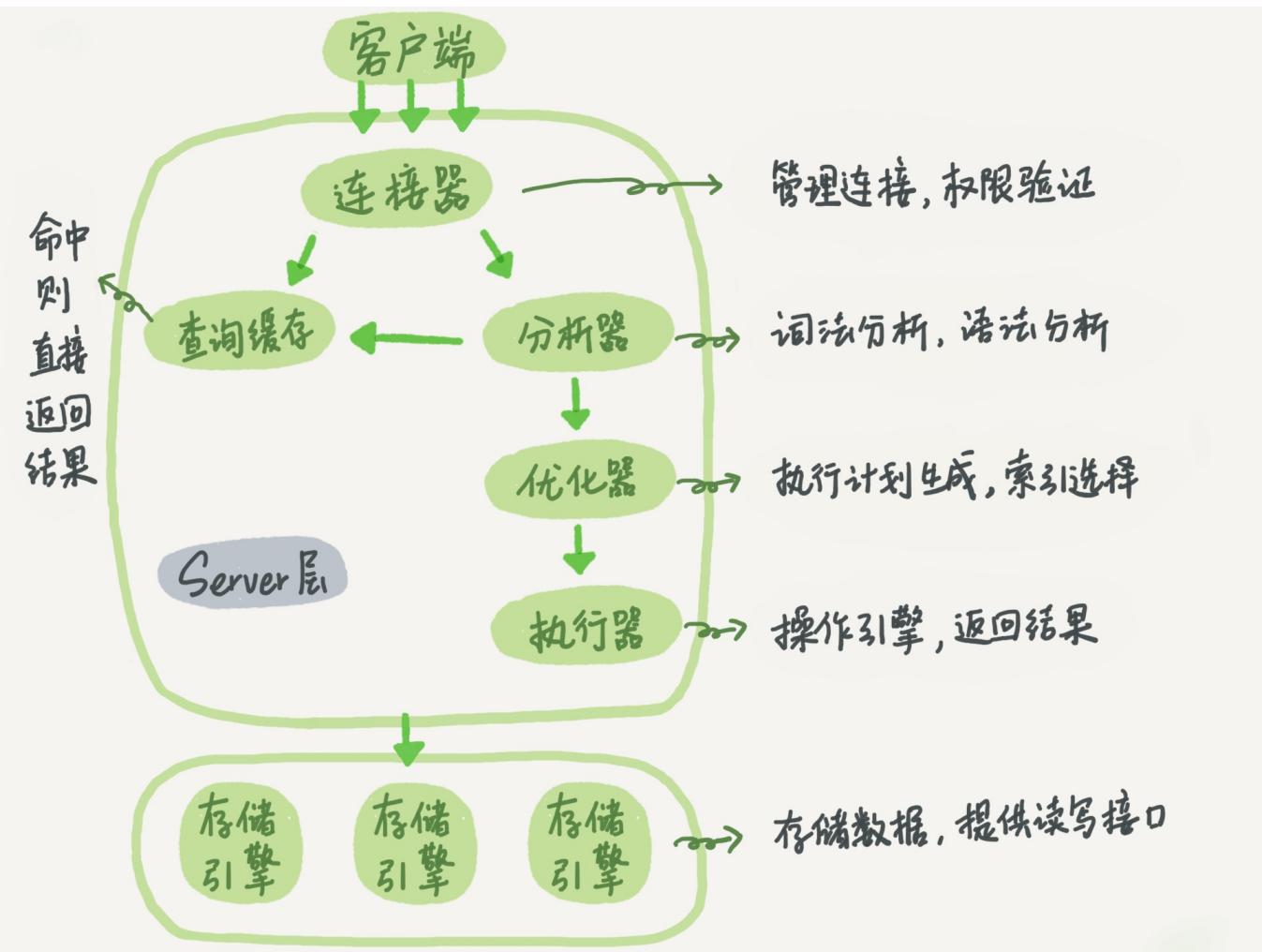
我们还是从一个表的一条更新语句说起，下面是这个表的创建语句，这个表有一个主键ID和一个整型字段C：

```
mysql> create table T(ID int primary key, c int);
```

如果要将ID=2这一行的值加1，SQL语句就会这么写：

```
mysql> update T set c=c+1 where ID=2;
```

前面我有跟你介绍过SQL语句基本的执行链路，这里我再把那张图拿过来，你也可以先简单看看这个图回顾下。首先，可以确定的说，查询语句的那一套流程，更新语句也是同样会走一遍。



MySQL的逻辑架构图

你执行语句前要先连接数据库，这是连接器的工作。

前面我们说过，在一个表上有更新的时候，跟这个表有关的查询缓存会失效，所以这条语句就会把表T上所有缓存结果都清空。这也就是我们一般不建议使用查询缓存的原因。

接下来，分析器会通过词法和语法解析知道这是一条更新语句。优化器决定要使用ID这个索引。然后，执行器负责具体执行，找到这一行，然后更新。

与查询流程不一样的是，更新流程还涉及两个重要的日志模块，它们正是我们今天要讨论的主角：**redo log**（重做日志）和 **binlog**（归档日志）。如果接触MySQL，那这两个词肯定是绕不过的，我后面的内容里也会不断地和你强调。不过话说回来，**redo log**和**binlog**在设计上有很多有意思的地方，这些设计思路也可以用到你自己的程序里。

重要的日志模块： **redo log**

不知道你还记不记得《孔乙己》这篇文章，酒店掌柜有一个粉板，专门用来记录客人的赊账记录。如果赊账的人不多，那么他可以把顾客名和账目写在板上。但如果赊账的人多了，粉板总会有记不下的时候，这个时候掌柜一定还有一个专门记录赊账的账本。

如果有人要赊账或者还账的话，掌柜一般有两种做法：

- 一种做法是直接把账本翻出来，把这次赊的账加上去或者扣除掉；
- 另一种做法是先在粉板上记下这次的账，等打烊以后再把账本翻出来核算。

在生意红火柜台很忙时，掌柜一定会选择后者，因为前者操作实在是太麻烦了。首先，你得找到这个人的赊账总额那条记录。你想想，密密麻麻几十页，掌柜要找到那个名字，可能还得带上老花镜慢慢找，找到之后再拿出算盘计算，最后再将结果写回到账本上。

这整个过程想想都麻烦。相比之下，还是先在粉板上记一下方便。你想想，如果掌柜没有粉板的帮助，每次记账都得翻账本，效率是不是低得让人难以忍受？

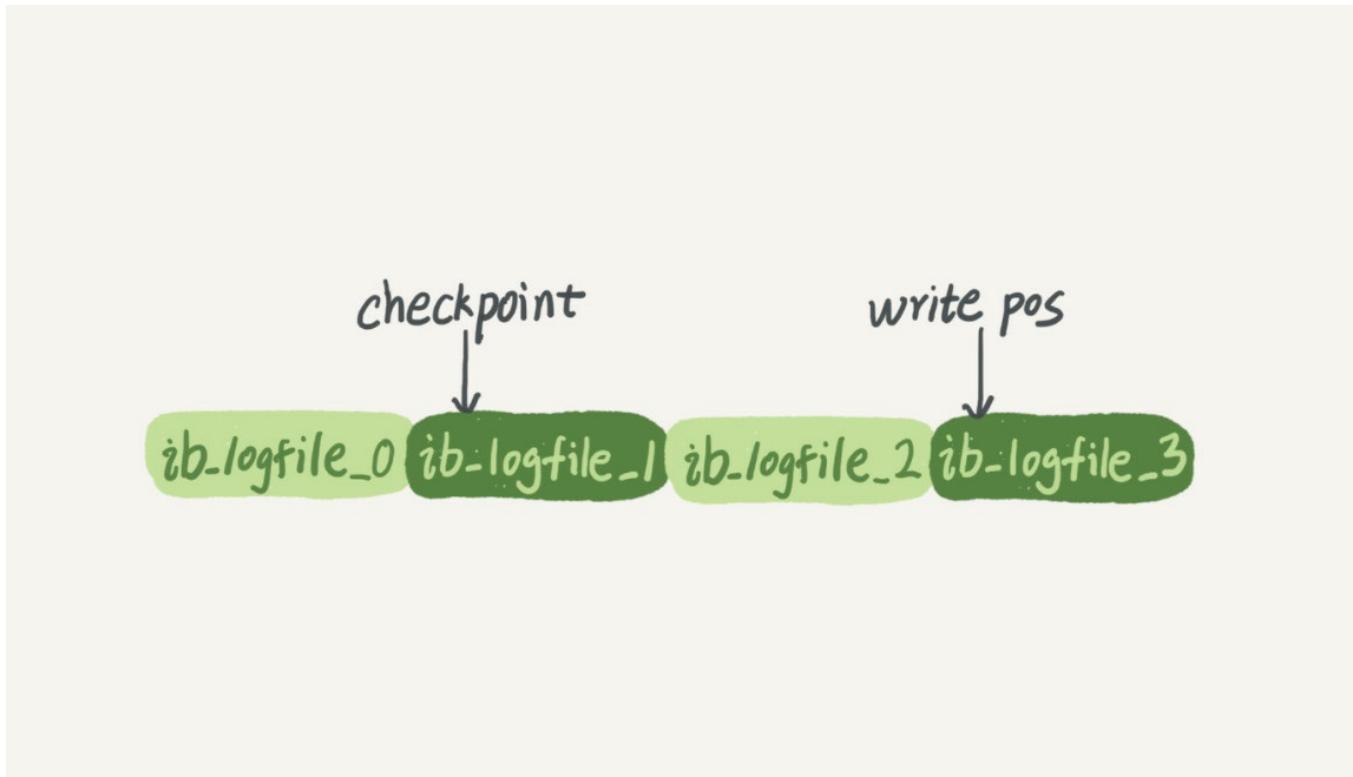
同样，在MySQL里也有这个问题，如果每一次的更新操作都需要写进磁盘，然后磁盘也要找到对应的那条记录，然后再更新，整个过程IO成本、查找成本都很高。为了解决这个问题，MySQL的设计者就用了类似酒店掌柜粉板的思路来提升更新效率。

而粉板和账本配合的整个过程，其实就是MySQL里经常说到的WAL技术，WAL的全称是Write-Ahead Logging，它的关键点就是先写日志，再写磁盘，也就是先写粉板，等不忙的时候再写账本。

具体来说，当有一条记录需要更新的时候，InnoDB引擎就会先把记录写到redo log（粉板）里面，并更新内存，这个时候更新就算完成了。同时，InnoDB引擎会在适当的时候，将这个操作记录更新到磁盘里面，而这个更新往往是在系统比较空闲的时候做，这就像打烊以后掌柜做的事。

如果今天赊账的不多，掌柜可以等打烊后再整理。但如果某天赊账的特别多，粉板写满了，又怎么办呢？这个时候掌柜只好放下手中的活儿，把粉板中的一部分赊账记录更新到账本中，然后把这些记录从粉板上擦掉，为记新账腾出空间。

与此类似，InnoDB的redo log是固定大小的，比如可以配置为一组4个文件，每个文件的大小是1GB，那么这块“粉板”总共就可以记录4GB的操作。从头开始写，写到末尾就又回到开头循环写，如下面这个图所示。



write pos是当前记录的位置，一边写一边后移，写到第3号文件末尾后就回到0号文件开头。
checkpoint是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到数据文件。

write pos和**checkpoint**之间的是“粉板”上还空着的部分，可以用来记录新的操作。如果**write pos**追上**checkpoint**，表示“粉板”满了，这时候不能再执行新的更新，得停下来先擦掉一些记录，把**checkpoint**推进一下。

有了**redo log**, InnoDB就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这个能力称为**crash-safe**。

要理解**crash-safe**这个概念，可以想想我们前面赊账记录的例子。只要赊账记录记在了粉板上或写在了账本上，之后即使掌柜忘记了，比如突然停业几天，恢复生意后依然可以通过账本和粉板上的数据明确赊账账目。

重要的日志模块： **binlog**

前面我们讲过，MySQL整体来看，其实就有两块：一块是**Server**层，它主要做的是MySQL功能层面的事情；还有一块是引擎层，负责存储相关的具体事宜。上面我们聊到的粉板**redo log**是InnoDB引擎特有的日志，而**Server**层也有自己的日志，称为**binlog**（归档日志）。

我想你肯定会问，为什么会有两份日志呢？

因为最开始MySQL里并没有InnoDB引擎。MySQL自带的引擎是MyISAM，但是MyISAM没有**crash-safe**的能力，**binlog**日志只能用于归档。而InnoDB是另一个公司以插件形式引入MySQL的，既然只依靠**binlog**是没有**crash-safe**能力的，所以InnoDB使用另外一套日志系统——也就是

redo log来实现**crash-safe**能力。

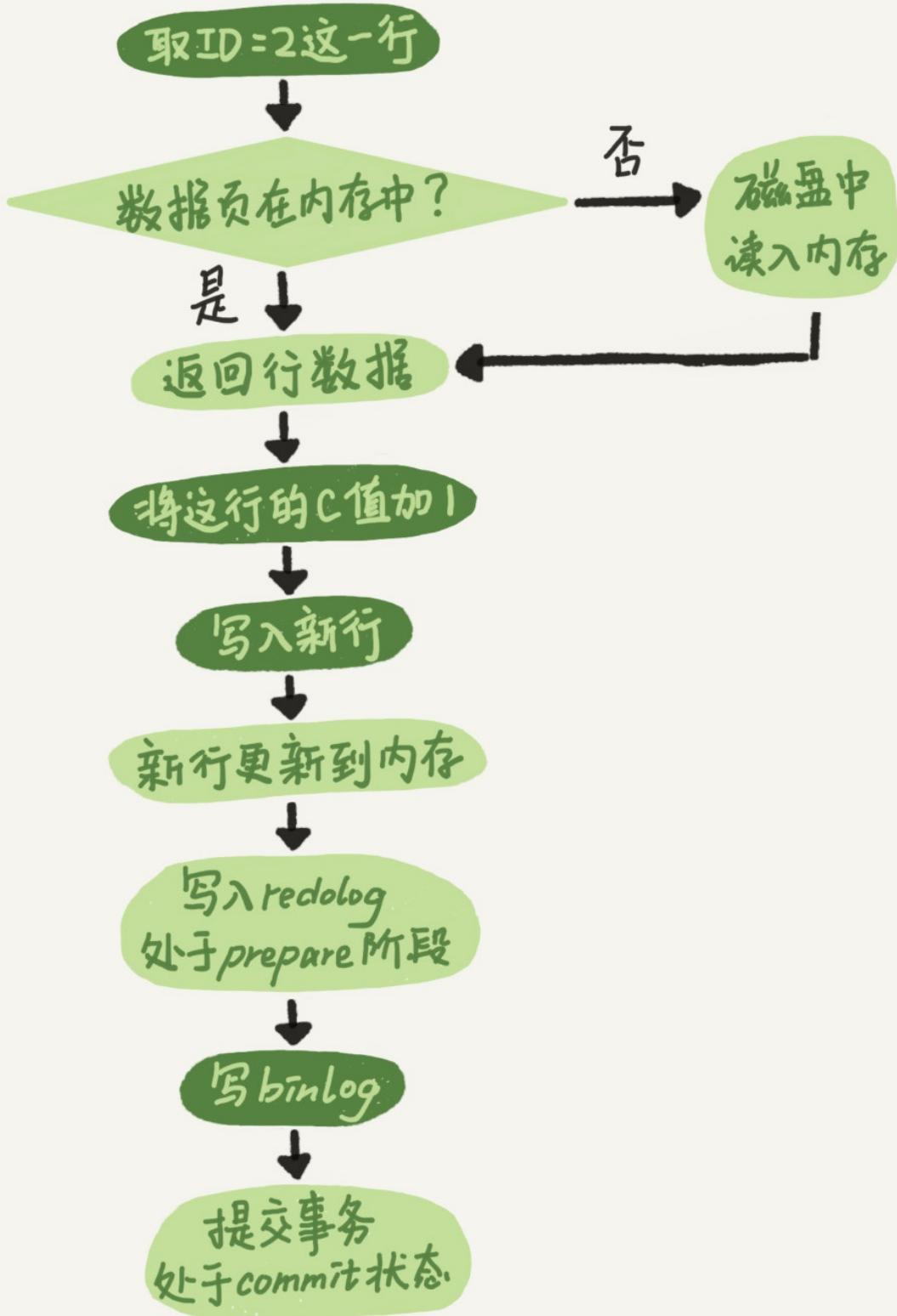
这两种日志有以下三点不同。

1. **redo log**是**InnoDB**引擎特有的；**binlog**是**MySQL**的**Server**层实现的，所有引擎都可以使用。
2. **redo log**是物理日志，记录的是“在某个数据页上做了什么修改”；**binlog**是逻辑日志，记录的是这个语句的原始逻辑，比如“给**ID=2**这一行的**c**字段加**1**”。
3. **redo log**是循环写的，空间固定会用完；**binlog**是可以追加写入的。“追加写”是指**binlog**文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

有了对这两个日志的概念性理解，我们再来看执行器和**InnoDB**引擎在执行这个简单的**update**语句时的内部流程。

1. 执行器先找引擎取**ID=2**这一行。**ID**是主键，引擎直接用树搜索找到这一行。如果**ID=2**这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。
2. 执行器拿到引擎给的行数据，把这个值加上**1**，比如原来是**N**，现在就是**N+1**，得到新的一行数据，再调用引擎接口写入这行新数据。
3. 引擎将这行新数据更新到内存中，同时将这个更新操作记录到**redo log**里面，此时**redo log**处于**prepare**状态。然后告知执行器执行完成了，随时可以提交事务。
4. 执行器生成这个操作的**binlog**，并把**binlog**写入磁盘。
5. 执行器调用引擎的提交事务接口，引擎把刚刚写入的**redo log**改成提交（**commit**）状态，更新完成。

这里我给出这个**update**语句的执行流程图，图中浅色框表示是在**InnoDB**内部执行的，深色框表示是在执行器中执行的。



update语句执行流程

你可能注意到了，最后三步看上去有点“绕”，将redo log的写入拆成了两个步骤：prepare和commit，这就是“两阶段提交”。

两阶段提交

为什么必须有“两阶段提交”呢？这是为了让两份日志之间的逻辑一致。要说明这个问题，我们得从文章开头的那个问题说起：怎样让数据库恢复到半个月内任意一秒的状态？

前面我们说过了，**binlog**会记录所有的逻辑操作，并且是采用“追加写”的形式。如果你的**DBA**承诺说半个月内可以恢复，那么备份系统中一定会保存最近半个月的所有**binlog**，同时系统会定期做整库备份。这里的“定期”取决于系统的重要性，可以是一天一备，也可以是一周一备。

当需要恢复到指定的某一秒时，比如某天下午两点发现中午十二点有一次误删表，需要找回数据，那你可以这么做：

- 首先，找到最近的一次全量备份，如果你运气好，可能就是昨天晚上的一个备份，从这个备份恢复到临时库；
- 然后，从备份的时间点开始，将备份的**binlog**依次取出来，重放到中午误删表之前的那个时刻。

这样你的临时库就跟误删之前的线上库一样了，然后你可以把表数据从临时库取出来，按需要恢复到线上库去。

好了，说完了数据恢复过程，我们回来说说，为什么日志需要“两阶段提交”。这里不妨用反证法来进行解释。

由于**redo log**和**binlog**是两个独立的逻辑，如果不采用两阶段提交，要么就是先写完**redo log**再写**binlog**，或者采用反过来的顺序。我们看看这两种方式会有什么问题。

仍然用前面的**update**语句来做例子。假设当前**ID=2**的行，字段**c**的值是**0**，再假设执行**update**语句过程中在写完第一个日志后，第二个日志还没有写完期间发生了**crash**，会出现什么情况呢？

1. 先写**redo log**后写**binlog**。假设在**redo log**写完，**binlog**还没有写完的时候，**MySQL**进程异常重启。由于我们前面说过的，**redo log**写完之后，系统即使崩溃，仍然能够把数据恢复回来，所以恢复后这一行**c**的值是**1**。

但是由于**binlog**没写完就**crash**了，这时候**binlog**里面就没有记录这个语句。因此，之后备份日志的时候，存起来的**binlog**里面就没有这条语句。

然后你会发现，如果需要用这个**binlog**来恢复临时库的话，由于这个语句的**binlog**丢失，这个临时库就会少了这一次更新，恢复出来的这一行**c**的值就是**0**，与原库的值不同。

2. 先写**binlog**后写**redo log**。如果在**binlog**写完之后**crash**，由于**redo log**还没写，崩溃恢复以后这个事务无效，所以这一行**c**的值是**0**。但是**binlog**里面已经记录了“把**c**从**0**改成**1**”这个日志。所以，在之后用**binlog**来恢复的时候就多了一个事务出来，恢复出来的这一行**c**的值就是**1**，与原库的值不同。

可以看到，如果不使用“两阶段提交”，那么数据库的状态就有可能和用它的日志恢复出来的库的状态不一致。

你可能会说，这个概率是不是很低，平时也没有什么动不动就需要恢复临时库的场景呀？

其实不是的，不只是误操作后需要用这个过程来恢复数据。当你需要扩容的时候，也就是需要再多搭建一些备库来增加系统的读能力的时候，现在常见的做法也是用全量备份加上应用binlog来实现的，这个“不一致”就会导致你的线上出现主从数据库不一致的情况。

简单说，**redo log**和**binlog**都可以用于表示事务的提交状态，而两阶段提交就是让这两个状态保持逻辑上的一致。

小结

今天，我介绍了MySQL里面最重要的两个日志，即物理日志**redo log**和逻辑日志**binlog**。

redo log用于保证crash-safe能力。`innodb_flush_log_at_trx_commit`这个参数设置成1的时候，表示每次事务的**redo log**都直接持久化到磁盘。这个参数我建议你设置成1，这样可以保证MySQL异常重启之后数据不丢失。

`sync_binlog`这个参数设置成1的时候，表示每次事务的**binlog**都持久化到磁盘。这个参数我也建议你设置成1，这样可以保证MySQL异常重启之后**binlog**不丢失。

我还跟你介绍了与MySQL日志系统密切相关的“两阶段提交”。两阶段提交是跨系统维持数据逻辑一致性时常用的一个方案，即使你不做数据库内核开发，日常开发中也有可能会用到。

文章的最后，我给你留一个思考题吧。前面我说到定期全量备份的周期“取决于系统重要性，有的是一天一备，有的是一周一备”。那么在什么场景下，一天一备会比一周一备更有优势呢？或者说，它影响了这个数据库系统的哪个指标？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾给出我的答案。

感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

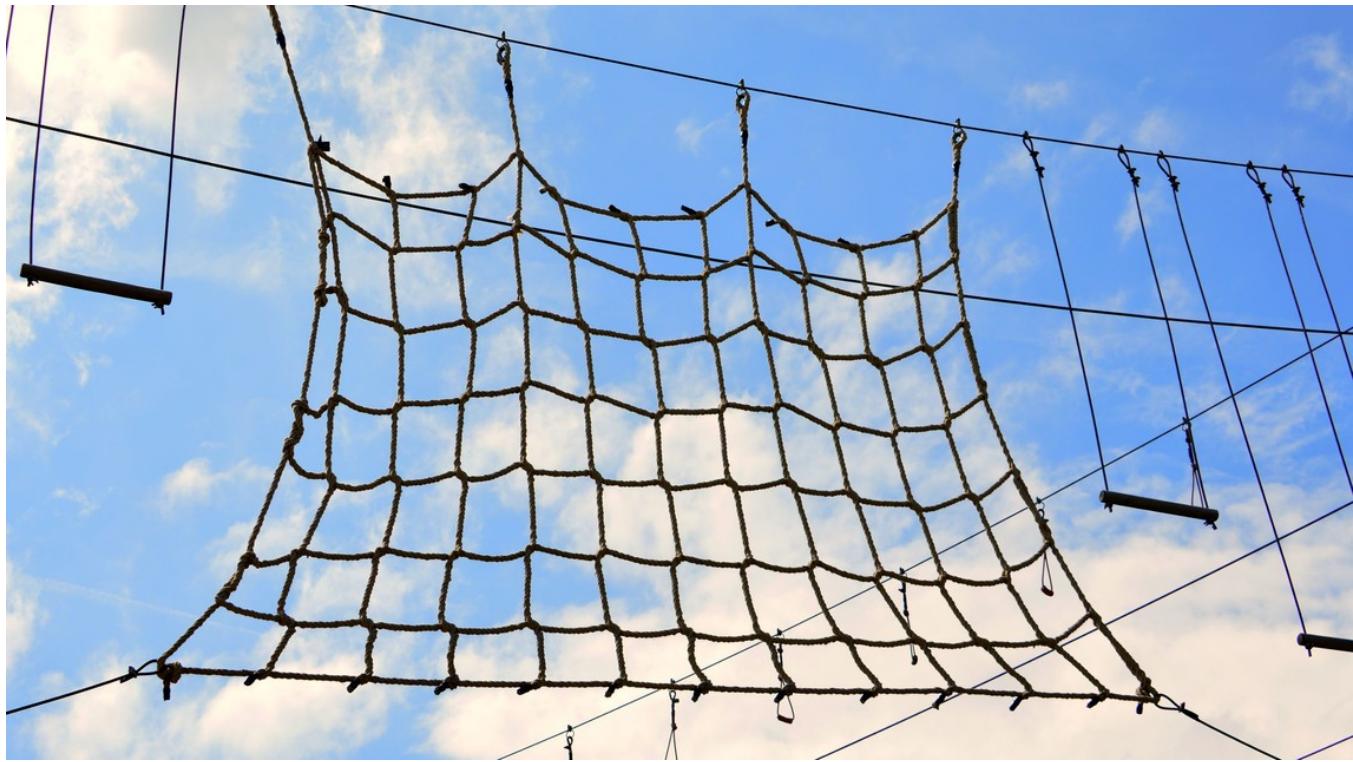
林晓斌

网名丁奇
前阿里资深技术专家



03 | 事务隔离：为什么你改了我还看不见？

2018-11-19 林晓斌



提到事务，你肯定不陌生，和数据库打交道的时候，我们总是会用到事务。最经典的例子就是转账，你要给朋友小王转100块钱，而此时你的银行卡只有100块钱。

转账过程具体到程序里会有一系列的操作，比如查询余额、做加减法、更新余额等，这些操作必须保证是一体的，不然等程序查完之后，还没做减法之前，你这100块钱，完全可以借着这个时间差再查一次，然后再给另外一个朋友转账，如果银行这么整，不就乱了么？这时就要用到“事务”这个概念了。

简单来说，事务就是要保证一组数据库操作，要么全部成功，要么全部失败。在MySQL中，事务支持是在引擎层实现的。你现在知道，MySQL是一个支持多引擎的系统，但并不是所有的引擎都支持事务。比如MySQL原生的MyISAM引擎就不支持事务，这也是MyISAM被InnoDB取代的重要原因之一。

今天的文章里，我将会以InnoDB为例，剖析MySQL在事务支持方面的特定实现，并基于原理给出相应的实践建议，希望这些案例能加深你对MySQL事务原理的理解。

隔离性与隔离级别

提到事务，你肯定会想到ACID（Atomicity、Consistency、Isolation、Durability，即原子性、一致性、隔离性、持久性），今天我们就来说说其中I，也就是“隔离性”。

当数据库上有多个事务同时执行的时候，就可能出现脏读（dirty read）、不可重复读（non-

`repeatable read`)、幻读(`phantom read`)的问题，为了解决这些问题，就有了“隔离级别”的概念。

在谈隔离级别之前，你首先要知道，你隔离得越严实，效率就会越低。因此很多时候，我们都要在二者之间寻找一个平衡点。**SQL**标准的事务隔离级别包括：读未提交(`read uncommitted`)、读提交(`read committed`)、可重复读(`repeatable read`)和串行化(`serializable`)。下面我逐一为你解释：

- 读未提交是指，一个事务还没提交时，它做的变更就能被别的事务看到。
- 读提交是指，一个事务提交之后，它做的变更才会被其他事务看到。
- 可重复读是指，一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。当然在可重复读隔离级别下，未提交变更对其他事务也是不可见的。
- 串行化，顾名思义是对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

其中“读提交”和“可重复读”比较难理解，所以我用一个例子说明这几种隔离级别。假设数据表T中只有一列，其中一行的值为1，下面是按照时间顺序执行两个事务的行为。

```
mysql> create table T(c int) engine=InnoDB;
insert into T(c) values(1);
```

事务A	事务B
启动事务 查询得到值1	启动事务
	查询得到值1
	将1改成2
查询得到值V1	
	提交事务B
查询得到值V2	
提交事务A	
查询得到值V3	

我们来看看在不同的隔离级别下，事务A会有哪些不同的返回结果，也就是图里面V1、V2、V3的返回值分别是什么。

- 若隔离级别是“读未提交”，则V1的值就是2。这时候事务B虽然还没有提交，但是结果已经被A看到了。因此，V2、V3也都是2。
- 若隔离级别是“读提交”，则V1是1，V2的值是2。事务B的更新在提交后才能被A看到。所以，V3的值也是2。

- 若隔离级别是“可重复读”，则V1、V2是1，V3是2。之所以V2还是1，遵循的就是这个要求：事务在执行期间看到的数据前后必须是一致的。
- 若隔离级别是“串行化”，则在事务B执行“将1改成2”的时候，会被锁住。直到事务A提交后，事务B才可以继续执行。所以从A的角度看，V1、V2值是1，V3的值是2。

在实现上，数据库里面会创建一个视图，访问的时候以视图的逻辑结果为准。在“可重复读”隔离级别下，这个视图是在事务启动时创建的，整个事务存在期间都用这个视图。在“读提交”隔离级别下，这个视图是在每个SQL语句开始执行的时候创建的。这里需要注意的是，“读未提交”隔离级别下直接返回记录上的最新值，没有视图概念；而“串行化”隔离级别下直接用加锁的方式来避免并行访问。

我们可以看到在不同的隔离级别下，数据库行为是有所不同的。Oracle数据库的默认隔离级别其实就是“读提交”，因此对于一些从Oracle迁移到MySQL的应用，为保证数据库隔离级别的一致，你一定要记得将MySQL的隔离级别设置为“读提交”。

配置的方式是，将启动参数transaction-isolation的值设置成READ-COMMITTED。你可以用show variables来查看当前的值。

```
mysql> show variables like 'transaction_isolation';
```

```
+-----+-----+
```

```
| Variable_name | Value |
```

```
+-----+-----+
```

```
| transaction_isolation | READ-COMMITTED |
```

```
+-----+-----+
```

总结来说，存在即合理，哪个隔离级别都有它自己的使用场景，你要根据自己的业务情况来定。我想你可能会问那什么时候需要“可重复读”的场景呢？我们来看一个数据校对逻辑的案例。

假设你在管理一个个人银行账户表。一个表存了每个月月底的余额，一个表存了账单明细。这时候你要做数据校对，也就是判断上个月的余额和当前余额的差额，是否与本月的账单明细一致。你一定希望在校对过程中，即使有用户发生了一笔新的交易，也不影响你的校对结果。

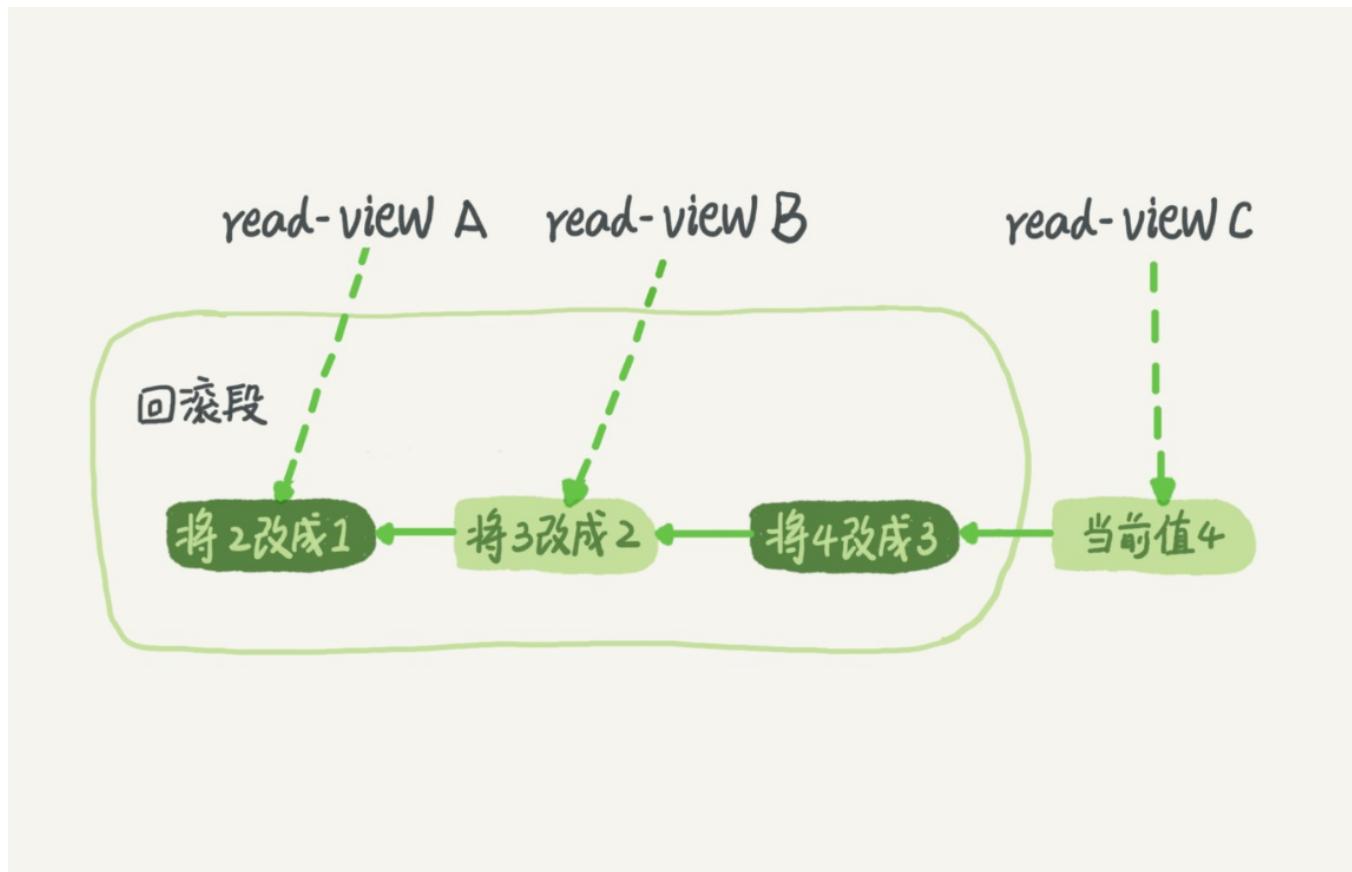
这时候使用“可重复读”隔离级别就很方便。事务启动时的视图可以认为是静态的，不受其他事务更新的影响。

事务隔离的实现

理解了事务的隔离级别，我们再来看看事务隔离具体是怎么实现的。这里我们展开说明“可重复读”。

在MySQL中，实际上每条记录在更新的时候都会同时记录一条回滚操作。记录上的最新值，通过回滚操作，都可以得到前一个状态的值。

假设一个值从1被按顺序改成了2、3、4，在回滚日志里面就会有类似下面的记录。



当前值是4，但是在查询这条记录的时候，不同时刻启动的事务会有不同的**read-view**。如图中看到的，在视图A、B、C里面，这一个记录的值分别是1、2、4，同一条记录在系统中可以存在多个版本，就是数据库的多版本并发控制（MVCC）。对于**read-view A**，要得到1，就必须将当前值依次执行图中所有的回滚操作得到。

同时你会发现，即使现在有另外一个事务正在将4改成5，这个事务跟**read-view A**、**B**、**C**对应的事务是不会冲突的。

你一定会问，回滚日志总不能一直保留吧，什么时候删除呢？答案是，在不需要的时候才删除。也就是说，系统会判断，当没有事务再需要用到这些回滚日志时，回滚日志会被删除。

什么时候才不需要了呢？就是当系统里没有比这个回滚日志更早的**read-view**的时候。

基于上面的说明，我们来讨论一下为什么建议你尽量不要使用长事务。

长事务意味着系统里面会存在很老的事务视图。由于这些事务随时可能访问数据库里面的任何数据，所以这个事务提交之前，数据库里面它可能用到的回滚记录都必须保留，这就会导致大量占用存储空间。

在MySQL 5.5及以前的版本，回滚日志是跟数据字典一起放在`ibdata`文件里的，即使长事务最终提交，回滚段被清理，文件也不会变小。我见过数据只有20GB，而回滚段有200GB的库。最终只好为了清理回滚段，重建整个库。

除了对回滚段的影响，长事务还占用锁资源，也可能拖垮整个库，这个我们会在后面讲锁的时候展开。

事务的启动方式

如前所述，长事务有这些潜在风险，我当然是建议你尽量避免。其实很多时候业务开发同学并不是有意使用长事务，通常是由于误用所致。MySQL的事务启动方式有以下几种：

1. 显式启动事务语句，`begin` 或 `start transaction`。配套的提交语句是`commit`，回滚语句是`rollback`。
2. `set autocommit=0`，这个命令会将这个线程的自动提交关掉。意味着如果你只执行一个`select`语句，这个事务就启动了，而且并不会自动提交。这个事务持续存在直到你主动执行`commit` 或 `rollback` 语句，或者断开连接。

有些客户端连接框架会默认连接成功后先执行一个`set autocommit=0`的命令。这就导致接下来的查询都在事务中，如果是长连接，就导致了意外的长事务。

因此，我会建议你总是使用`set autocommit=1`，通过显式语句的方式来启动事务。

但是有的开发同学会纠结“多一次交互”的问题。对于一个需要频繁使用事务的业务，第二种方式每个事务在开始时都不需要主动执行一次“`begin`”，减少了语句的交互次数。如果你也有这个顾虑，我建议你使用`commit work and chain`语法。

在`autocommit`为1的情况下，用`begin`显式启动的事务，如果执行`commit`则提交事务。如果执行`commit work and chain`，则是提交事务并自动启动下一个事务，这样也省去了再次执行`begin`语句的开销。同时带来的好处是从程序开发的角度明确地知道每个语句是否处于事务中。

你可以在`information_schema`库的`innodb_trx`这个表中查询长事务，比如下面这个语句，用于查找持续时间超过60s的事务。

```
select * from information_schema.innodb_trx where TIME_TO_SEC(timediff(now(),trx_started))>60
```

小结

这篇文章里面，我介绍了MySQL的事务隔离级别的现象和实现，根据实现原理分析了长事务存在的风险，以及如何用正确的方式避免长事务。希望我举的例子能够帮助你理解事务，并更好地使用MySQL的事务特性。

我给你留一个问题吧。你现在知道了系统里面应该避免长事务，如果你是业务开发负责人同时也是数据库负责人，你会有什么方案来避免出现或者处理这种情况呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

在上期文章的最后，我给你留下的问题是“一天一备跟一周一备的对比”。

好处是“最长恢复时间”更短。

在一天一备的模式里，最坏情况下需要应用一天的binlog。比如，你每天0点做一次全量备份，而要恢复出一个到昨天晚上23点的备份。

一周一备最坏情况就要应用一周的binlog了。

系统的对应指标就是 @尼古拉斯·赵四 @慕塔 提到的RTO（恢复目标时间）。

当然这个是有成本的，因为更频繁全量备份需要消耗更多存储空间，所以这个RTO是成本换来的，就需要你根据业务重要性来评估了。

同时也感谢 @super blue cat、@高枕、@Jason 留下了高质量的评论。



04 | 深入浅出索引（上）

2018-11-21 林晓斌



提到数据库索引，我想你并不陌生，在日常工作中会经常接触到。比如某一个SQL查询比较慢，分析完原因之后，你可能就会说“给某个字段加个索引吧”之类的解决方案。但到底什么是索引，索引又是如何工作的呢？今天就让我们一起来聊聊这个话题吧。

数据库索引的内容比较多，我分成了上下两篇文章。索引是数据库系统里面最重要的概念之一，所以我希望你能够耐心看完。在后面的实战文章中，我也会经常引用这两篇文章中提到的知识点，加深你对数据库索引的理解。

一句话简单来说，**索引的出现其实就是为了提高数据查询的效率，就像书的目录一样**。一本500页的书，如果你想快速找到其中的某一个知识点，在不借助目录的情况下，那我估计你可得找一会儿。同样，对于数据库的表而言，索引其实就是它的“目录”。

索引的常见模型

索引的出现是为了提高查询效率，但是实现索引的方式却有很多种，所以这里也就引入了索引模型的概念。可以用于提高读写效率的数据结构很多，这里我先给你介绍三种常见、也比较简单的数据结构，它们分别是哈希表、有序数组和搜索树。

下面我主要从使用的角度，为你简单分析一下这三种模型的区别。

哈希表是一种以键-值（key-value）存储数据的结构，我们只要输入待查找的值即**key**，就可以找到其对应的值即**Value**。哈希的思路很简单，把值放在数组里，用一个哈希函数把**key**换算成一个

确定的位置，然后把value放在数组的这个位置。

不可避免地，多个key值经过哈希函数的换算，会出现同一个值的情况。处理这种情况的一种方法是，拉出一个链表。

假设，你现在维护着一个身份证信息和姓名的表，需要根据身份证号查找对应的名字，这时对应的哈希索引的示意图如下所示：

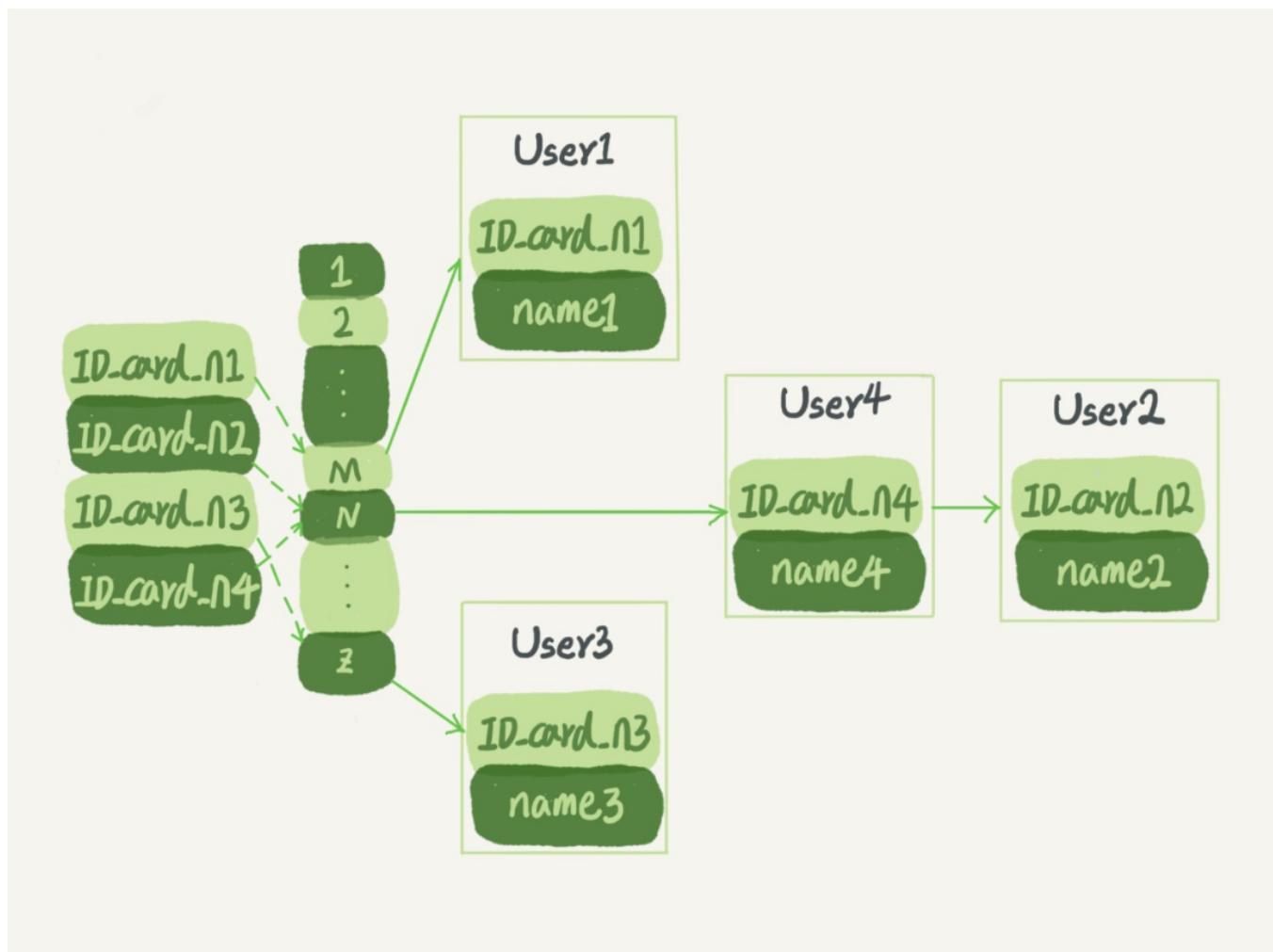


图1 哈希表示意图

图中，User2和User4根据身份证号算出来的值都是N，但没关系，后面还跟了一个链表。假设，这时候你要查ID_card_n2对应的名字是什么，处理步骤就是：首先，将ID_card_n2通过哈希函数算出N；然后，按顺序遍历，找到User2。

需要注意的是，图中四个ID_card_n的值并不是递增的，这样做的好处是增加新的User时速度会很快，只需要往后追加。但缺点是，因为不是有序的，所以哈希索引做区间查询的速度是很慢的。

你可以设想下，如果你现在要找身份证号在[ID_card_X, ID_card_Y]这个区间的所有的用户，就必须全部扫描一遍了。

所以，哈希表这种结构适用于只有等值查询的场景，比如Memcached及其他一些NoSQL引

擎。

而有序数组在等值查询和范围查询场景中的性能就都非常优秀。还是上面这个根据身份证号查名字的例子，如果我们使用有序数组来实现的话，示意图如下所示：

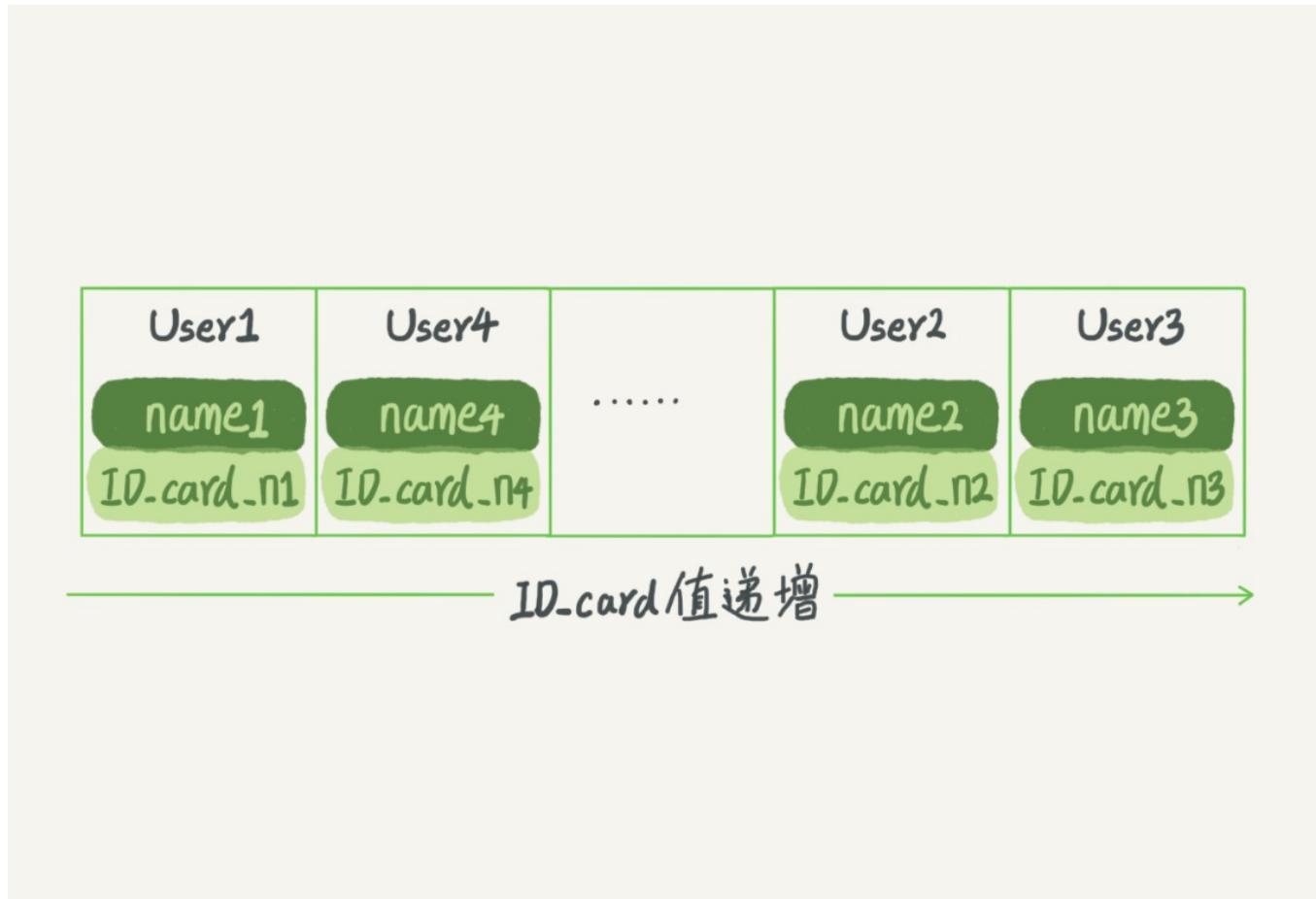


图2 有序数组示意图

这里我们假设身份证号没有重复，这个数组就是按照身份证号递增的顺序保存的。这时候如果你要查ID_card_n2对应的名字，用二分法就可以快速得到，这个时间复杂度是 $O(\log(N))$ 。

同时很显然，这个索引结构支持范围查询。你要查身份证号在[ID_card_X, ID_card_Y]区间的User，可以先用二分法找到ID_card_X（如果不存在ID_card_X，就找到大于ID_card_X的第一个User），然后向右遍历，直到查到第一个大于ID_card_Y的身份证号，退出循环。

如果仅仅看查询效率，有序数组就是最好的数据结构了。但是，在需要更新数据的时候就麻烦了，你往中间插入一个记录就必须得挪动后面所有的记录，成本太高。

所以，有序数组索引只适用于静态存储引擎，比如你要保存的是2017年某个城市的所有人口信息，这类不会再修改的数据。

二叉搜索树也是课本里的经典数据结构了。还是上面根据身份证号查名字的例子，如果我们用二叉搜索树来实现的话，示意图如下所示：

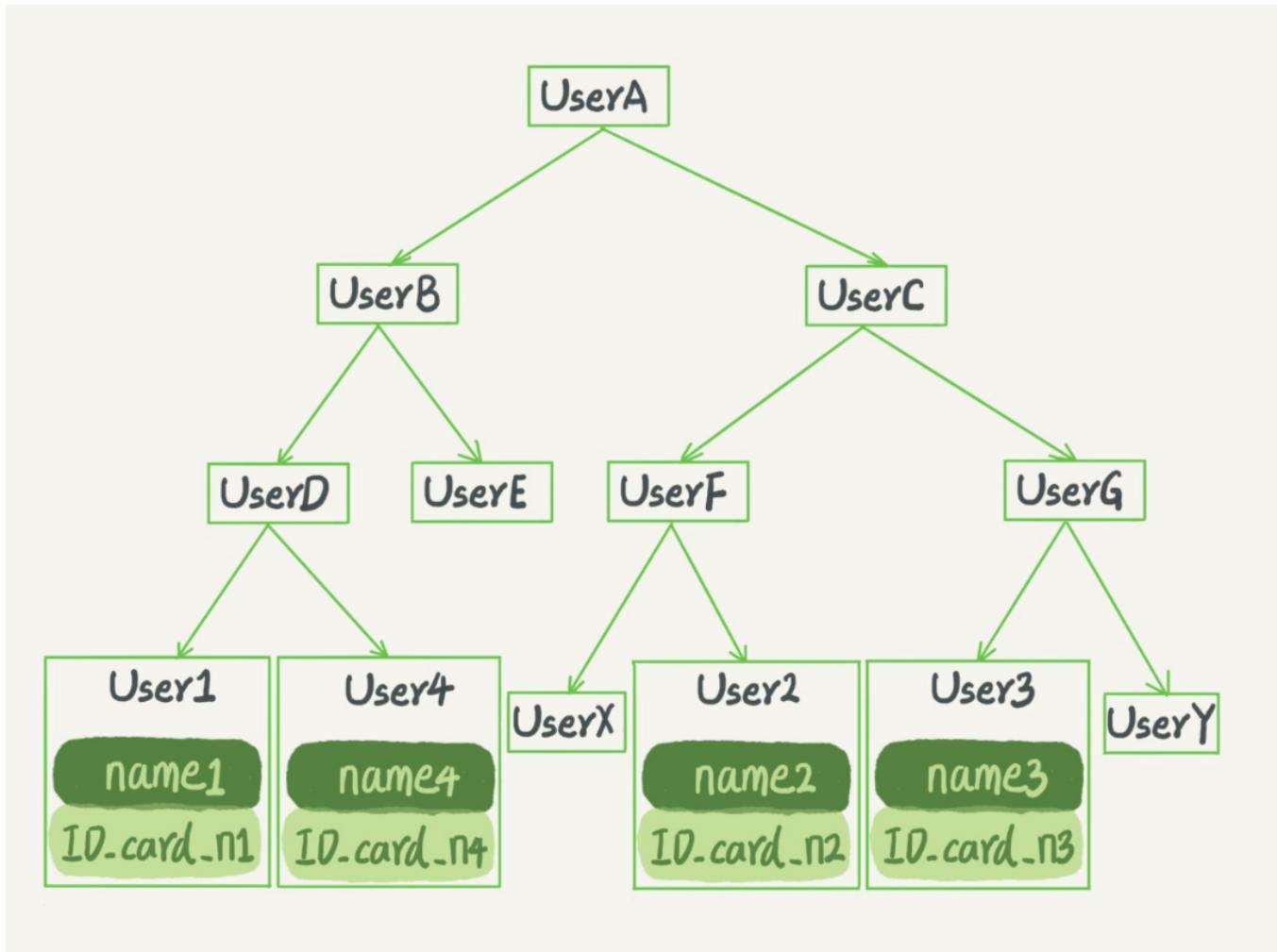


图3 二叉搜索树示意图

二叉搜索树的特点是：每个节点的左儿子小于父节点，父节点又小于右儿子。这样如果你要查 `ID_card_n2` 的话，按照图中的搜索顺序就是按照 `UserA -> UserC -> UserF -> User2` 这个路径得到。这个时间复杂度是 $O(\log(N))$ 。

当然为了维持 $O(\log(N))$ 的查询复杂度，你就需要保持这棵树是平衡二叉树。为了做这个保证，更新的时间复杂度也是 $O(\log(N))$ 。

树可以有二叉，也可以有多叉。多叉树就是每个节点有多个儿子，儿子之间的大小保证从左到右递增。二叉树是搜索效率最高的，但是实际上大多数的数据存储却并不使用二叉树。其原因是，索引不止存在内存中，还要写到磁盘上。

你可以想象下一棵 100 万节点的平衡二叉树，树高 20。一次查询可能需要访问 20 个数据块。在机械硬盘时代，从磁盘随机读一个数据块需要 10 ms 左右的寻址时间。也就是说，对于一个 100 万行的表，如果使用二叉树来存储，单独访问一个行可能需要 20 个 10 ms 的时间，这个查询可真够慢的。

为了让一个查询尽量少地读磁盘，就必须让查询过程访问尽量少的数据块。那么，我们就不应该使用二叉树，而是要使用“N 叉”树。这里，“N 叉”树中的“N”取决于数据块的大小。

以InnoDB的一个整数字段索引为例，这个N差不多是1200。这棵树高是4的时候，就可以存1200的3次方个值，这已经17亿了。考虑到树根的数据块总是在内存中的，一个10亿行的表上一个整数字段的索引，查找一个值最多只需要访问3次磁盘。其实，树的第二层也有很大概率在内存中，那么访问磁盘的平均次数就更少了。

N叉树由于在读写上的性能优点，以及适配磁盘的访问模式，已经被广泛应用在数据库引擎中了。

不管是哈希还是有序数组，或者N叉树，它们都是不断迭代、不断优化的产物或者解决方案。数据库技术发展到今天，跳表、LSM树等数据结构也被用于引擎设计中，这里我就不再一一展开了。

你心里要有个概念，数据库底层存储的核心就是基于这些数据模型的。每碰到一个新数据库，我们需要先关注它的数据模型，这样才能从理论上分析出这个数据库的适用场景。

截止到这里，我用了半篇文章的篇幅和你介绍了不同的数据结构，以及它们的适用场景，你可能会觉得有些枯燥。但是，我建议你还是要多花一些时间来理解这部分内容，毕竟这是数据库处理数据的核心概念之一，在分析问题的时候会经常用到。当你理解了索引的模型后，就会发现在分析问题的时候会有一个更清晰的视角，体会到引擎设计的精妙之处。

现在，我们一起进入相对偏实战的内容吧。

在MySQL中，索引是在存储引擎层实现的，所以并没有统一的索引标准，即不同存储引擎的索引的工作方式并不一样。而即使多个存储引擎支持同一种类型的索引，其底层的实现也可能不同。由于InnoDB存储引擎在MySQL数据库中使用最为广泛，所以下面我就以InnoDB为例，和你分析一下其中的索引模型。

InnoDB 的索引模型

在InnoDB中，表都是根据主键顺序以索引的形式存放的，这种存储方式的表称为索引组织表。又因为前面我们提到的，InnoDB使用了B+树索引模型，所以数据都是存储在B+树中的。

每一个索引在InnoDB里面对应一棵B+树。

假设，我们有一个主键列为ID的表，表中有字段k，并且在k上有索引。

这个表的建表语句是：

```

mysql> create table T(
    id int primary key,
    k int not null,
    name varchar(16),
    index (k))engine=InnoDB;

```

表中R1~R5的(ID,k)值分别为(100,1)、(200,2)、(300,3)、(500,5)和(600,6)，两棵树的示例示意图如下。

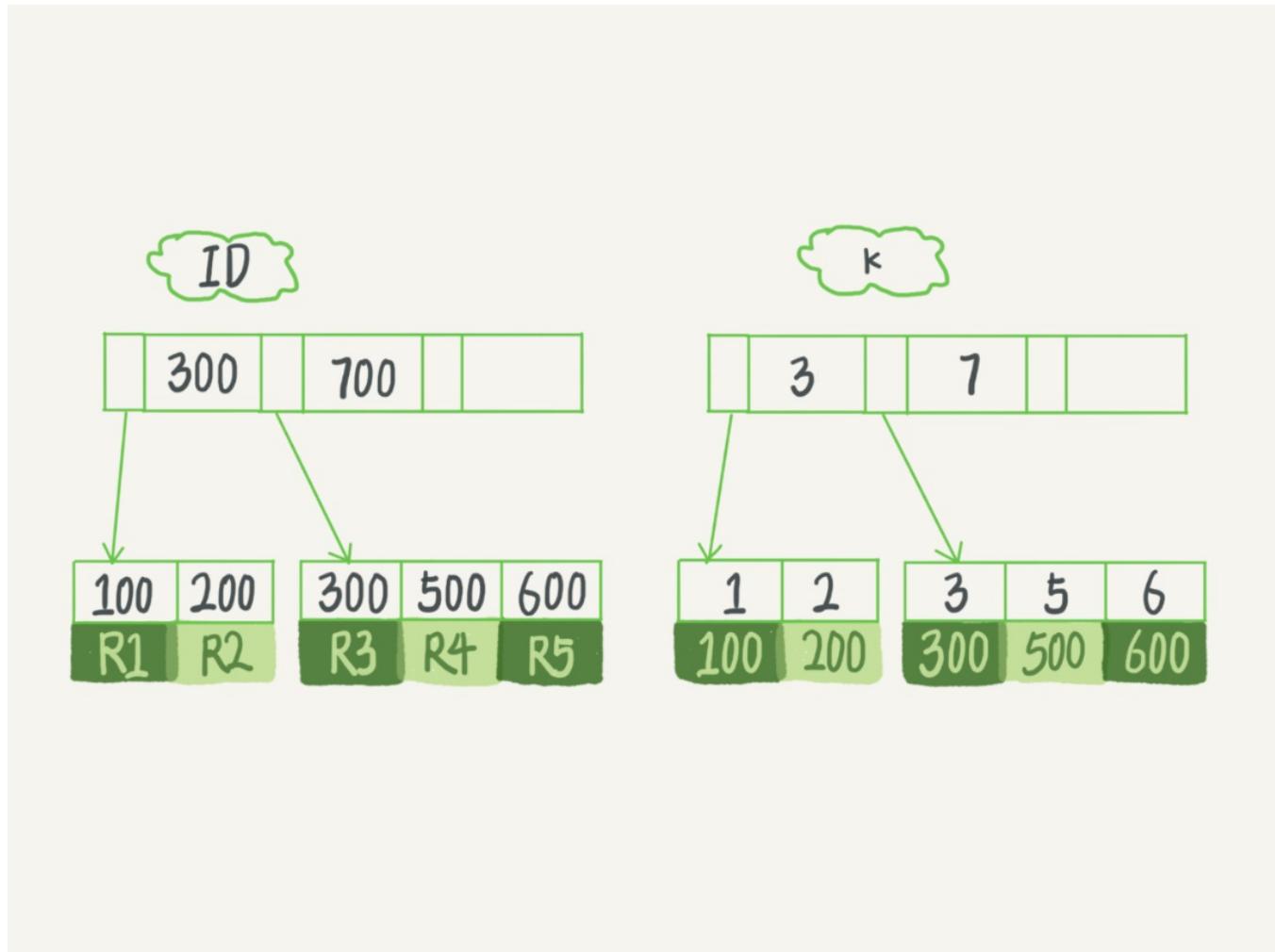


图4 InnoDB的索引组织结构

从图中不难看出，根据叶子节点的内容，索引类型分为主键索引和非主键索引。

主键索引的叶子节点存的是整行数据。在InnoDB里，主键索引也被称为聚簇索引（**clustered index**）。

非主键索引的叶子节点内容是主键的值。在InnoDB里，非主键索引也被称为二级索引（**secondary index**）。

根据上面的索引结构说明，我们来讨论一个问题：基于主键索引和普通索引的查询有什么区

别？

- 如果语句是`select * from T where ID=500`, 即主键查询方式, 则只需要搜索ID这棵B+树;
- 如果语句是`select * from T where k=5`, 即普通索引查询方式, 则需要先搜索k索引树, 得到ID的值为500, 再到ID索引树搜索一次。这个过程称为回表。

也就是说, 基于非主键索引的查询需要多扫描一棵索引树。因此, 我们在应用中应该尽量使用主键查询。

索引维护

B+树为了维护索引有序性, 在插入新值的时候需要做必要的维护。以上面这个图为例, 如果插入新的行ID值为700, 则只需要在R5的记录后面插入一个新记录。如果新插入的ID值为400, 就相对麻烦了, 需要逻辑上挪动后面的数据, 空出位置。

而更糟的情况是, 如果R5所在的数据页已经满了, 根据B+树的算法, 这时候需要申请一个新的数据页, 然后挪动部分数据过去。这个过程称为页分裂。在这种情况下, 性能自然会受影响。

除了性能外, 页分裂操作还影响数据页的利用率。原本放在一个页的数据, 现在分到两个页中, 整体空间利用率降低大约50%。

当然有分裂就有合并。当相邻两个页由于删除了数据, 利用率很低之后, 会将数据页做合并。合并的过程, 可以认为是分裂过程的逆过程。

基于上面的索引维护过程说明, 我们来讨论一个案例:

你可能在一些建表规范里面见到过类似的描述, 要求建表语句里一定要有自增主键。当然事无绝对, 我们来分析一下哪些场景下应该使用自增主键, 而哪些场景下不应该。

自增主键是指自增列上定义的主键, 在建表语句中一般是这么定义的: `NOT NULL PRIMARY KEY AUTO_INCREMENT`。

插入新记录的时候可以不指定ID的值, 系统会获取当前ID最大值加1作为下一条记录的ID值。

也就是说, 自增主键的插入数据模式, 正符合了我们前面提到的递增插入的场景。每次插入一条新记录, 都是追加操作, 都不涉及到挪动其他记录, 也不会触发叶子节点的分裂。

而有业务逻辑的字段做主键, 则往往不容易保证有序插入, 这样写数据成本相对较高。

除了考虑性能外, 我们还可以从存储空间的角度来看。假设你的表中确实有一个唯一字段, 比如字符串类型的身份证号, 那应该用身份证号做主键, 还是用自增字段做主键呢?

由于每个非主键索引的叶子节点上都是主键的值。如果用身份证号做主键, 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。

显然，主键长度越小，普通索引的叶子节点就越小，普通索引占用的空间也就越小。

所以，从性能和存储空间方面考量，自增主键往往是更合理的选择。

有没有什么场景适合用业务字段直接做主键的呢？还是有的。比如，有些业务的场景需求是这样的：

1. 只有一个索引；
2. 该索引必须是唯一索引。

你一定看出来了，这就是典型的**KV**场景。

由于没有其他索引，所以也就不用考虑其他索引的叶子节点大小的问题。

这时候我们就要优先考虑上一段提到的“尽量使用主键查询”原则，直接将这个索引设置为主键，可以避免每次查询需要搜索两棵树。

小结

今天，我跟你分析了数据库引擎可用的数据结构，介绍了**InnoDB**采用的**B+**树结构，以及为什么**InnoDB**要这么选择。**B+**树能够很好地配合磁盘的读写特性，减少单次查询的磁盘访问次数。

由于**InnoDB**是索引组织表，一般情况下我会建议你创建一个自增主键，这样非主键索引占用的空间最小。但事无绝对，我也跟你讨论了使用业务逻辑字段做主键的应用场景。

最后，我给你留下一个问题吧。对于上面例子中的**InnoDB**表T，如果你要重建索引 k，你的两个SQL语句可以这么写：

```
alter table T drop index k;  
alter table T add index(k);
```

如果你要重建主键索引，也可以这么写：

```
alter table T drop primary key;  
alter table T add primary key(id);
```

我的问题是，对于上面这两个重建索引的作法，说出你的理解。如果有不合适的，为什么，更好的方法是什么？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾给出我的参考答案。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章末尾给你留下的问题是：如何避免长事务对业务的影响？

这个问题，我们可以从应用开发端和数据库端来看。

首先，从应用开发端来看：

1. 确认是否使用了`set autocommit=0`。这个确认工作可以在测试环境中开展，把MySQL的`general_log`开起来，然后随便跑一个业务逻辑，通过`general_log`的日志来确认。一般框架如果会设置这个值，也就会提供参数来控制行为，你的目标就是把它改成1。
2. 确认是否有不必要的只读事务。有些框架会习惯不管什么语句先用`begin/commit`框起来。我见过有些是业务并没有这个需要，但是也好几个`select`语句放到了事务中。这种只读事务可以去掉。
3. 业务连接数据库的时候，根据业务本身的预估，通过`SET MAX_EXECUTION_TIME`命令，来控制每个语句执行的最长时间，避免单个语句意外执行太长时间。（为什么会意外？在后续的文章中会提到这类案例）

其次，从数据库端来看：

1. 监控`information_schema.Innodb_trx`表，设置长事务阈值，超过就报警/或者`kill`；
2. Percona的`pt-kill`这个工具不错，推荐使用；
3. 在业务功能测试阶段要求输出所有的`general_log`，分析日志行为提前发现问题；
4. 如果使用的是MySQL 5.6或者更新版本，把`innodb_undo_tablespaces`设置成2（或更大的值）。如果真的出现大事务导致回滚段过大，这样设置后清理起来更方便。

感谢 @壹笙@漂泊 @王凯 @易翔 留下的高质量评论。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



05 | 深入浅出索引（下）

2018-11-23 林晓斌



在上一篇文章中，我和你介绍了InnoDB索引的数据结构模型，今天我们再继续聊聊跟MySQL索引有关的概念。

在开始这篇文章之前，我们先来看一下这个问题：

在下面这个表T中，如果我执行 `select * from T where k between 3 and 5`，需要执行几次树的搜索操作，会扫描多少行？

下面是这个表的初始化语句。

```
mysql> create table T (
    ID int primary key,
    k int NOT NULL DEFAULT 0,
    s varchar(16) NOT NULL DEFAULT '',
    index k(k)
    engine=InnoDB;

insert into T values(100,1, 'aa'),(200,2, 'bb'),(300,3, 'cc'),(500,5, 'ee'),(600,6, 'ff'),(700,7, 'gg');
```

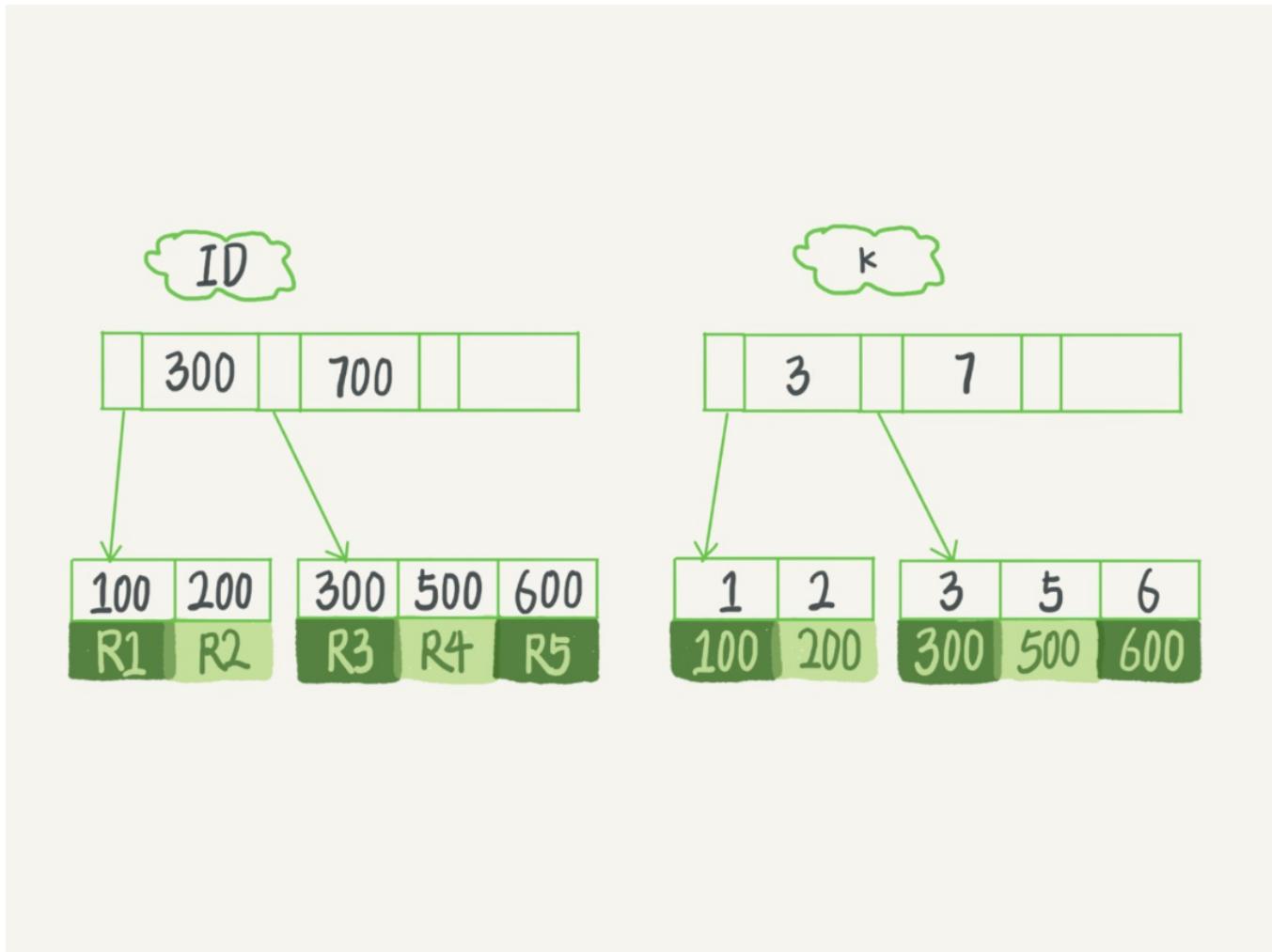


图1 InnoDB的索引组织结构

现在，我们一起来看看这条SQL查询语句的执行流程：

1. 在k索引树上找到k=3的记录，取得 ID = 300;
2. 再到ID索引树查到ID=300对应的R3;
3. 在k索引树取下一个值k=5，取得ID=500;
4. 再回到ID索引树查到ID=500对应的R4;
5. 在k索引树取下一个值k=6，不满足条件，循环结束。

在这个过程中，回到主键索引树搜索的过程，我们称为回表。可以看到，这个查询过程读了k索引树的3条记录（步骤1、3和5），回表了两次（步骤2和4）。

在这个例子中，由于查询结果所需要的数据只在主键索引上有，所以不得不回表。那么，有没有可能经过索引优化，避免回表过程呢？

覆盖索引

如果执行的语句是select ID from T where k between 3 and 5，这时只需要查ID的值，而ID的值

已经在**k**索引树上了，因此可以直接提供查询结果，不需要回表。也就是说，在这个查询里面，索引**k**已经“覆盖了”我们的查询需求，我们称为覆盖索引。

由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段。

需要注意的是，在引擎内部使用覆盖索引在索引**k**上其实读了三个记录，**R3~R5**（对应的索引**k**上的记录项），但是对于MySQL的**Server**层来说，它就是找引擎拿到了两条记录，因此MySQL认为扫描行数是**2**。

备注：关于如何查看扫描行数的问题，我将会在第16文章《如何正确地显示随机消息？》中，和你详细讨论。

基于上面覆盖索引的说明，我们来讨论一个问题：在一个市民信息表上，是否有必要将身份证号和名字建立联合索引？

假设这个市民表的定义是这样的：

```
CREATE TABLE `tuser` (
  `id` int(11) NOT NULL,
  `id_card` varchar(32) DEFAULT NULL,
  `name` varchar(32) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `ismale` tinyint(1) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `id_card` (`id_card`),
  KEY `name_age` (`name`, `age`)
) ENGINE=InnoDB
```

我们知道，身份证号是市民的唯一标识。也就是说，如果有根据身份证号查询市民信息的需求，我们只要在身份证号字段上建立索引就够了。而再建立一个（身份证号、姓名）的联合索引，是不是浪费空间？

如果现在有一个高频请求，要根据市民的身份证号查询他的姓名，这个联合索引就有意义了。它可以在高频请求上用到覆盖索引，不再需要回表查整行记录，减少语句的执行时间。

当然，索引字段的维护总是有代价的。因此，在建立冗余索引来支持覆盖索引时就需要权衡考虑了。这正是业务**DBA**，或者称为业务数据架构师的工作。

最左前缀原则

看到这里你一定有一个疑问，如果为每一种查询都设计一个索引，索引是不是太多了。如果我现在要按照市民的身份证号去查他的家庭地址呢？虽然这个查询需求在业务中出现的概率不高，但总不能让它走全表扫描吧？反过来说，单独为一个不频繁的请求创建一个（身份证号，地址）的索引又感觉有点浪费。应该怎么做呢？

这里，我先和你说结论吧。**B+树**这种索引结构，可以利用索引的“最左前缀”，来定位记录。

为了直观地说明这个概念，我们用（name, age）这个联合索引来分析。

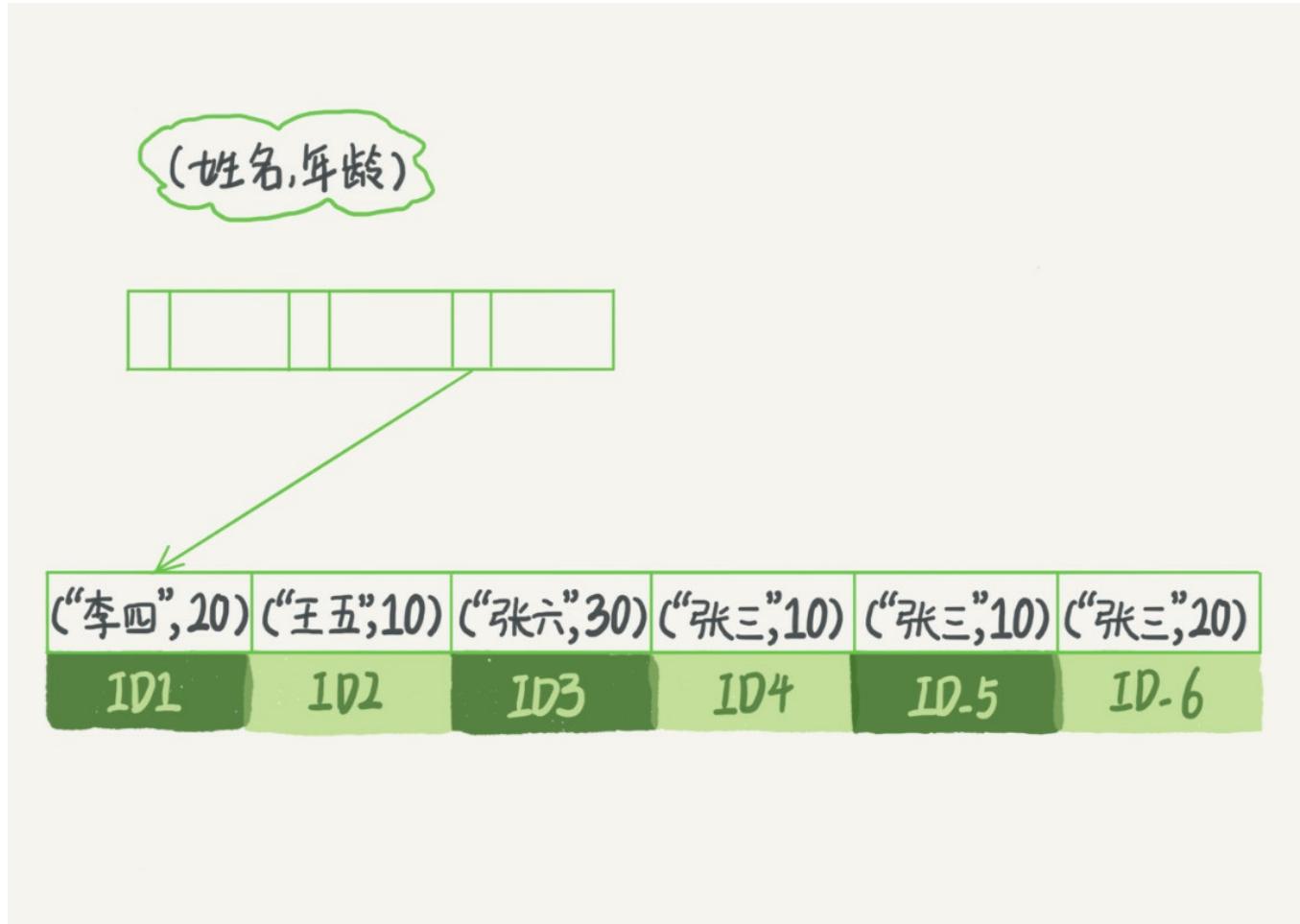


图2 (name, age) 索引示意图

可以看到，索引项是按照索引定义里面出现的字段顺序排序的。

当你的逻辑需求是查到所有名字是“张三”的人时，可以快速定位到ID4，然后向后遍历得到所有需要的结果。

如果你要查的是所有名字第一个字是“张”的人，你的SQL语句的条件是"where name like '张%'”。这时，你也能够用上这个索引，查找到第一个符合条件的记录是ID3，然后向后遍历，直到不满足条件为止。

可以看到，不只是索引的全部定义，只要满足最左前缀，就可以利用索引来加速检索。这个最左前缀可以是联合索引的最左N个字段，也可以是字符串索引的最左M个字符。

基于上面对最左前缀索引的说明，我们来讨论一个问题：在建立联合索引的时候，如何安排索引内的字段顺序。

这里我们的评估标准是，索引的复用能力。因为可以支持最左前缀，所以当已经有了(a,b)这个联合索引后，一般就不需要单独在a上建立索引了。因此，第一原则是，如果通过调整顺序，可以少维护一个索引，那么这个顺序往往就是需要优先考虑采用的。

所以现在你知道了，这段开头的问题里，我们要为高频请求创建(身份证号，姓名)这个联合索引，并用这个索引支持“根据身份证号查询地址”的需求。

那么，如果既有联合查询，又有基于a、b各自的查询呢？查询条件里面只有b的语句，是无法使用(a,b)这个联合索引的，这时候你不得不维护另外一个索引，也就是说你需要同时维护(a,b)、(b)这两个索引。

这时候，我们要考虑的原则就是空间了。比如上面这个市民表的情况，name字段是比age字段大的，那我就建议你创建一个(name,age)的联合索引和一个(age)的单字段索引。

索引下推

上一段我们说到满足最左前缀原则的时候，最左前缀可以用于在索引中定位记录。这时，你可能要问，那些不符合最左前缀的部分，会怎么样呢？

我们还是以市民表的联合索引(name, age)为例。如果现在有一个需求：检索出表中“名字第一个字是张，而且年龄是10岁的所有男孩”。那么，SQL语句是这么写的：

```
mysql> select * from tuser where name like '张%' and age=10 and ismale=1;
```

你已经知道了前缀索引规则，所以这个语句在搜索索引树的时候，只能用“张”，找到第一个满足条件的记录ID3。当然，这还不错，总比全表扫描要好。

然后呢？

当然是判断其他条件是否满足。

在MySQL 5.6之前，只能从ID3开始一个个回表。到主键索引上找出数据行，再对比字段值。

而MySQL 5.6引入的索引下推优化(index condition pushdown)，可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

图3和图4，是这两个过程的执行流程图。

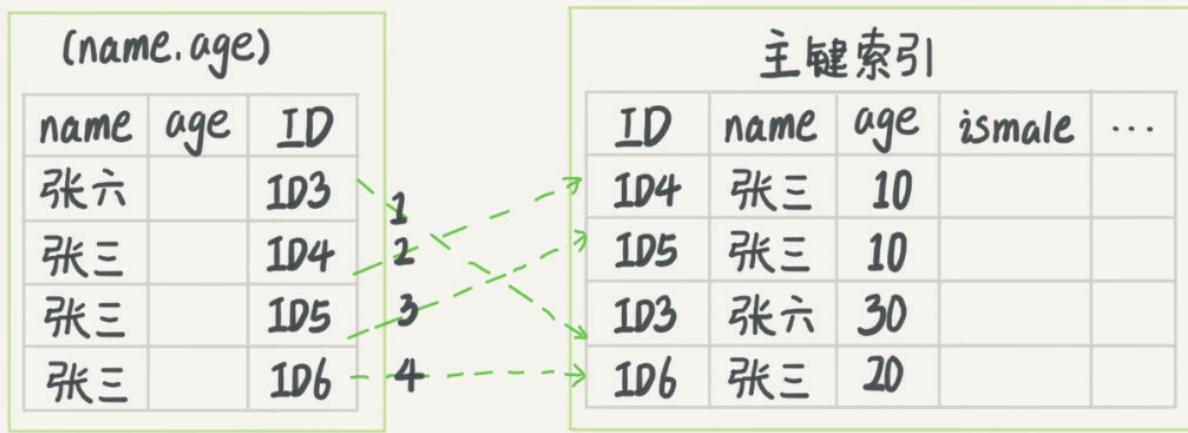


图3 无索引下推执行流程

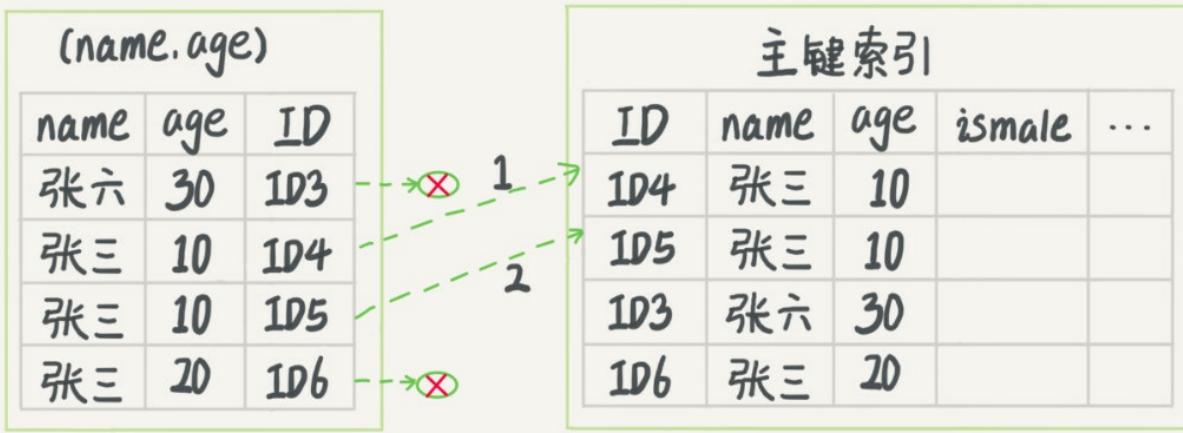


图4 索引下推执行流程

在图3和4这两个图里面，每一个虚线箭头表示回表一次。

图3中，在(name,age)索引里面我特意去掉了age的值，这个过程InnoDB并不会去看age的值，只是按顺序把“name第一个字是’张’”的记录一条条取出来回表。因此，需要回表4次。

图4跟图3的区别是，InnoDB在(name,age)索引内部就判断了age是否等于10，对于不等于10的记录，直接判断并跳过。在我们的这个例子中，只需要对ID4、ID5这两条记录回表取数据判断，就只需要回表2次。

小结

今天这篇文章，我和你继续讨论了数据库索引的概念，包括了覆盖索引、前缀索引、索引下推。你可以看到，在满足语句需求的情况下，尽量少地访问资源是数据库设计的重要原则之一。我们在使用数据库的时候，尤其是在设计表结构时，也要以减少资源消耗作为目标。

接下来我给你留下一个问题吧。

实际上主键索引也是可以使用多个字段的。DBA小吕在入职新公司的时候，就发现自己接手维护的库里面，有这么一个表，表结构定义类似这样的：

```
CREATE TABLE `geek` (
    `a` int(11) NOT NULL,
    `b` int(11) NOT NULL,
    `c` int(11) NOT NULL,
    `d` int(11) NOT NULL,
    PRIMARY KEY (`a`, `b`),
    KEY `c` (`c`),
    KEY `ca` (`c`, `a`),
    KEY `cb` (`c`, `b`)
) ENGINE=InnoDB;
```

公司的同事告诉他说，由于历史原因，这个表需要**a**、**b**做联合主键，这个小吕理解了。

但是，学过本章内容的小吕又纳闷了，既然主键包含了**a**、**b**这两个字段，那意味着单独在字段**c**上创建一个索引，就已经包含了三个字段了呀，为什么要创建“**ca**”“**cb**”这两个索引？

同事告诉他，是因为他们的业务里面有这样的两种语句：

```
select * from geek where c=N order by a limit 1;
select * from geek where c=N order by b limit 1;
```

我给你的问题是，这位同事的解释对吗，为了这两个查询模式，这两个索引是否都是必须的？为什么呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，通过两个`alter`语句重建索引**k**，以及通过两个`alter`语句重建主键索引是否合理。

在评论区，有同学问到为什么要重建索引。我们文章里面有提到，索引可能因为删除，或者页分裂等原因，导致数据页有空洞，重建索引的过程会创建一个新的索引，把数据按顺序插入，这样页面的利用率最高，也就是索引更紧凑、更省空间。

这道题目，我给你的“参考答案”是：

重建索引**k**的做法是合理的，可以达到省空间的目的。但是，重建主键的过程不合理。不论是删除主键还是创建主键，都会将整个表重建。所以连着执行这两个语句的话，第一个语句就白做了。这两个语句，你可以用这个语句代替：`alter table T engine=InnoDB`。在专栏的第12篇文章

《为什么表数据删掉一半，表文件大小不变？》中，我会和你分析这条语句的执行流程。

评论区留言中，@壹笙漂泊 做了很详细的笔记，@高枕 帮同学解答了问题，@约书亚 提了一个很不错的面试问题。在这里，我要和你们道一声感谢。

PS：如果你在面试中，曾有过被MySQL相关问题难住的经历，也可以把这个问题发到评论区，我们一起来讨论。如果解答这个问题，需要的篇幅会很长的话，我可以放到答疑文章展开。

The image is a promotional graphic for a MySQL course. It features a portrait of a man with glasses and short dark hair, wearing a black button-down shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45 讲' in large, bold, dark font. Below the title is a subtitle '从原理到实战，丁奇带你搞懂 MySQL' in a smaller, lighter font. In the top left corner, there is the logo for '极客时间' (Geek Time) which consists of a stylized orange 'G' icon followed by the text '极客时间'. At the bottom left, the author's name '林晓斌' is displayed in a large, dark font, with '网名丁奇' and '前阿里资深技术专家' in a smaller, lighter font below it.

06 | 全局锁和表锁：给表加个字段怎么有这么多阻碍？

2018-11-26 林晓斌



今天我要跟你聊聊MySQL的锁。数据库锁设计的初衷是处理并发问题。作为多用户共享的资源，当出现并发访问的时候，数据库需要合理地控制资源的访问规则。而锁就是用来实现这些访问规则的重要数据结构。

根据加锁的范围，MySQL里面的锁大致可以分成全局锁、表级锁和行锁三类。今天这篇文章，我会和你分享全局锁和表级锁。而关于行锁的内容，我会留着在下一篇文章中再和你详细介绍。

这里需要说明的是，锁的设计比较复杂，这两篇文章不会涉及锁的具体实现细节，主要介绍的是碰到锁时的现象和其背后的原理。

全局锁

顾名思义，全局锁就是对整个数据库实例加锁。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。

全局锁的典型使用场景是，做全库逻辑备份。也就是把整库每个表都select出来存成文本。

以前有一种做法，是通过**FTWRL**确保不会有其他线程对数据库做更新，然后对整个库做备份。注意，在备份过程中整个库完全处于只读状态。

但是让整库都只读，听上去就很危险：

- 如果你在主库上备份，那么在备份期间都不能执行更新，业务基本上就得停摆；
- 如果你在从库上备份，那么备份期间从库不能执行主库同步过来的binlog，会导致主从延迟。

看来加全局锁不太好。但是细想一下，备份为什么要加锁呢？我们来看一下不加锁会有什么问题。

假设你现在要维护“极客时间”的购买系统，关注的是用户账户余额表和用户课程表。

现在发起一个逻辑备份。假设备份期间，有一个用户，他购买了一门课程，业务逻辑里就要扣掉他的余额，然后往已购课程里面加上一门课。

如果时间顺序上是先备份账户余额表(`u_account`)，然后用户购买，然后备份用户课程表(`u_course`)，会怎么样呢？你可以看一下这个图：

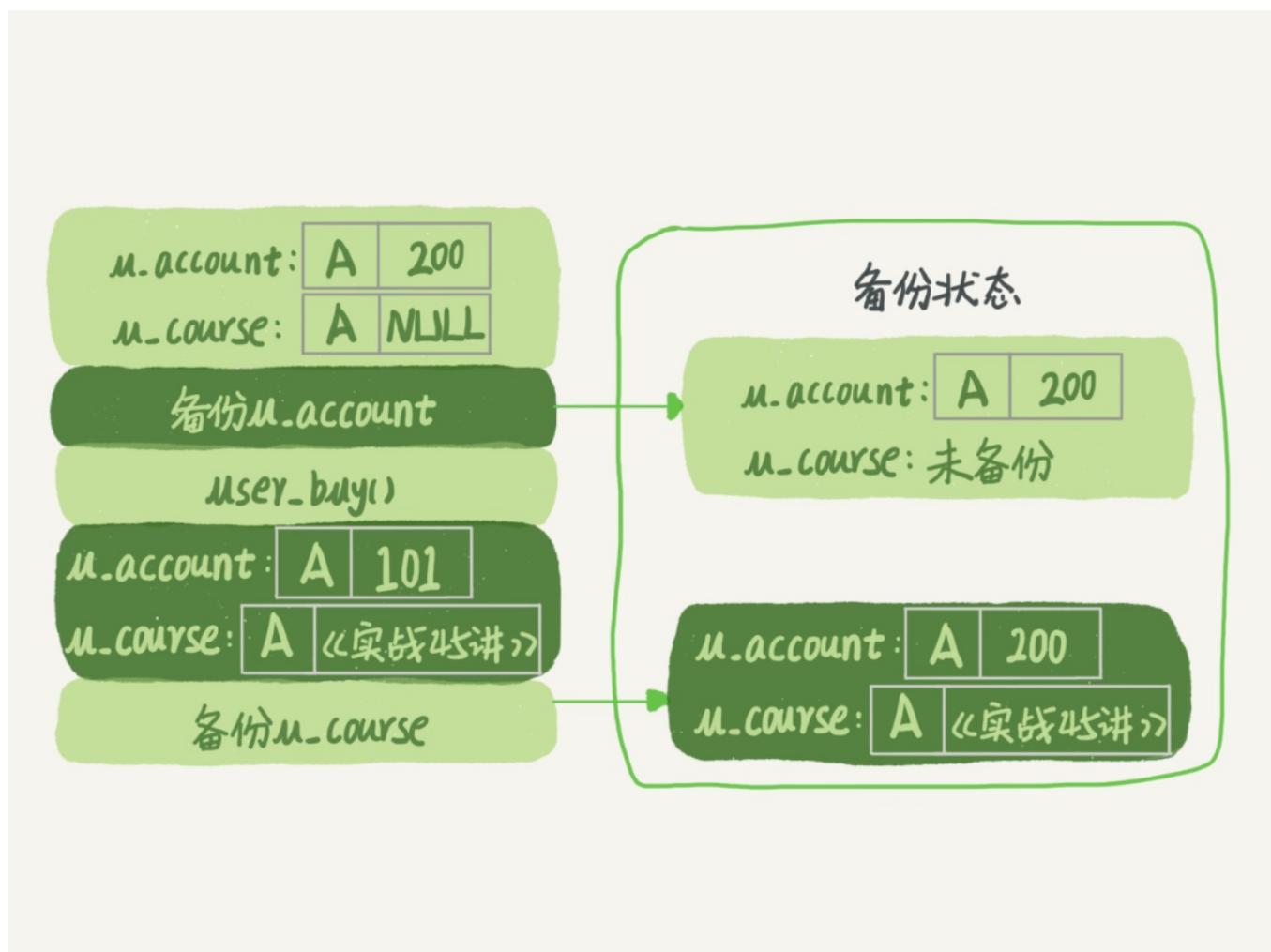


图1 业务和备份状态图

可以看到，这个备份结果里，用户A的数据状态是“账户余额没扣，但是用户课程表里面已经多了一门课”。如果后面用这个备份来恢复数据的话，用户A就发现，自己赚了。

作为用户可别觉得这样可真好啊，你可以试想一下：如果备份表的顺序反过来，先备份用户课程

表再备份账户余额表，又可能会出现什么结果？

也就是说，不加锁的话，备份系统备份的得到的库不是一个逻辑时间点，这个视图是逻辑不一致的。

说到视图你肯定想起来了，我们在前面讲事务隔离的时候，其实是一个方法能够拿到一致性视图的，对吧？

是的，就是在可重复读隔离级别下开启一个事务。

备注：如果你对事务隔离级别的概念不是很清晰的话，可以再回顾一下第3篇文章[《事务隔离：为什么你改了我还看不见？》](#)中的相关内容。

官方自带的逻辑备份工具是`mysqldump`。当`mysqldump`使用参数`-single-transaction`的时候，导数据之前就会启动一个事务，来确保拿到一致性视图。而由于MVCC的支持，这个过程中数据是可以正常更新的。

你一定在疑惑，有了这个功能，为什么还需要`FTWRL`呢？**一致性读是好，但前提是引擎要支持这个隔离级别。**比如，对于`MyISAM`这种不支持事务的引擎，如果备份过程中有更新，总是只能取到最新的数据，那么就破坏了备份的一致性。这时，我们就需要使用`FTWRL`命令了。

所以，`single-transaction`方法只适用于所有的表使用事务引擎的库。如果有的表使用了不支持事务的引擎，那么备份就只能通过`FTWRL`方法。这往往是DBA要求业务开发人员使用`InnoDB`替代`MyISAM`的原因之一。

你也许会问，既然要全库只读，为什么不使用`set global readonly=true`的方式呢？确实`readonly`方式也可以让全库进入只读状态，但我还是会建议你用`FTWRL`方式，主要有两个原因：

- 一是，在有些系统中，`readonly`的值会被用来做其他逻辑，比如用来判断一个库是主库还是备库。因此，修改`global`变量的方式影响面更大，我不建议你使用。
- 二是，在异常处理机制上有差异。如果执行`FTWRL`命令之后由于客户端发生异常断开，那么`MySQL`会自动释放这个全局锁，整个库回到可以正常更新的状态。而将整个库设置为`readonly`之后，如果客户端发生异常，则数据库就会一直保持`readonly`状态，这样会导致整个库长时间处于不可写状态，风险较高。

业务的更新不只是增删改数据（DML），还有可能是加字段等修改表结构的操作（DDL）。不论是哪种方法，一个库被全局锁上以后，你要对里面任何一个表做加字段操作，都是会被锁住的。

但是，即使没有被全局锁住，加字段也不是就能一帆风顺的，因为你还会碰到接下来我们要介绍的表级锁。

表级锁

MySQL里面表级别的锁有两种：一种是表锁，一种是元数据锁（meta data lock，MDL）。

表锁的语法是 **lock tables ...read/write**。与FTWRL类似，可以用**unlock tables**主动释放锁，也可以在客户端断开的时候自动释放。需要注意，**lock tables**语法除了会限制别的线程的读写外，也限定了本线程接下来的操作对象。

举个例子，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。

在还没有出现更细粒度的锁的时候，表锁是最常用的处理并发的方式。而对于InnoDB这种支持行锁的引擎，一般不使用**lock tables**命令来控制并发，毕竟锁住整个表的影响面还是太大。

另一类表级的锁是**MDL**（**metadata lock**）。MDL不需要显式使用，在访问一个表的时候会被自动加上。MDL的作用是，保证读写的正确性。你可以想象一下，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个表结构做变更，删了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

因此，在MySQL 5.5版本中引入了**MDL**，当对一个表做增删改查操作的时候，加**MDL**读锁；当要对表做结构变更操作的时候，加**MDL**写锁。

- 读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。
- 读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性。因此，如果有两个线程要同时给一个表加字段，其中一个要等另一个执行完才能开始执行。

虽然**MDL**锁是系统默认会加的，但却是你不能忽略的一个机制。比如下面这个例子，我经常看到有人掉到这个坑里：给一个小表加个字段，导致整个库挂了。

你肯定知道，给一个表加字段，或者修改字段，或者加索引，需要扫描全表的数据。在对大表操作的时候，你肯定会特别小心，以免对线上服务造成影响。而实际上，即使是小表，操作不慎也会出问题。我们来看一下下面的操作序列，假设表t是一个小表。

备注：这里的实验环境是MySQL 5.6。

session A	session B	session C	session D
begin; select * from t limit 1;			
	select * from t limit 1;		
		alter table t add f int; (blocked)	
			select * from t limit 1; (blocked)

我们可以看到**session A**先启动，这时候会对表t加一个**MDL**读锁。由于**session B**需要的也是**MDL**读锁，因此可以正常执行。

之后**session C**会被**blocked**，是因为**session A**的**MDL**读锁还没有释放，而**session C**需要**MDL**写锁，因此只能被阻塞。

如果只有**session C**自己被阻塞还没什么关系，但是之后所有要在表t上新申请**MDL**读锁的请求也会被**session C**阻塞。前面我们说了，所有对表的增删改查操作都需要先申请**MDL**读锁，就都被锁住，等于这个表现在完全不可读写了。

如果某个表上的查询语句频繁，而且客户端有重试机制，也就是说超时后会再起一个新**session**再请求的话，这个库的线程很快就会爆满。

你现在应该知道了，事务中的**MDL**锁，在语句执行开始时申请，但是语句结束后并不会马上释放，而会等到整个事务提交后再释放。

基于上面的分析，我们来讨论一个问题，如何安全地给小表加字段？

首先我们要解决长事务，事务不提交，就会一直占着**MDL**锁。在MySQL的**information_schema**库的 **innodb_trx** 表中，你可以查到当前执行中的事务。如果你要做**DDL**变更的表刚好有长事务在执行，要考虑先暂停**DDL**，或者**kill**掉这个长事务。

但考虑一下这个场景。如果你要变更的表是一个热点表，虽然数据量不大，但是上面的请求很频繁，而你不得不加个字段，你该怎么做呢？

这时候`kill`可能未必管用，因为新的请求马上就来了。比较理想的机制是，在`alter table`语句里面设定等待时间，如果在这个指定的等待时间里面能够拿到`MDL`写锁最好，拿不到也不要阻塞后面的业务语句，先放弃。之后开发人员或者`DBA`再通过重试命令重复这个过程。

`MariaDB`已经合并了`AliSQL`的这个功能，所以这两个开源分支目前都支持`DDL NOWAIT/WAIT n`这个语法。

```
ALTER TABLE tbl_name NOWAIT add column ...
ALTER TABLE tbl_name WAIT N add column ...
```

小结

今天，我跟你介绍了`MySQL`的全局锁和表级锁。

全局锁主要用在逻辑备份过程中。对于全部是`InnoDB`引擎的库，我建议你选择使用`-single-transaction`参数，对应用会更友好。

表锁一般是在数据库引擎不支持行锁的时候才会被用到的。如果你发现你的应用程序里有`lock tables`这样的语句，你需要追查一下，比较可能的情况是：

- 要么是你的系统现在还在用`MyISAM`这类不支持事务的引擎，那要安排升级换引擎；
- 要么是你的引擎升级了，但是代码还没升级。我见过这样的情况，最后业务开发就是把`lock tables`和`unlock tables`改成`begin`和`commit`，问题就解决了。

`MDL`会直到事务提交才释放，在做表结构变更的时候，你一定要小心不要导致锁住线上查询和更新。

最后，我给你留一个问题吧。备份一般都会在备库上执行，你在用`-single-transaction`方法做逻辑备份的过程中，如果主库上的一个小表做了一个`DDL`，比如给一个表上加了一列。这时候，从备库上会看到什么现象呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

说明：这篇文章没有介绍到物理备份，物理备份会有一篇单独的文章。

上期问题时间

上期的问题是关于对联合主键索引和`InnoDB`索引组织表的理解。

我直接贴@老杨同志 的回复略作修改如下（我修改的部分用橙色标出）：

表记录

-a--|-b--|-c--|-d--

123 d

132 d

143 d

213 d

222 d

234 d

主键 a, b的聚簇索引组织顺序相当于 **order by a,b** , 也就是先按a排序，再按b排序， c无序。

索引 ca 的组织是先按c排序，再按a排序，同时记录主键

-c--|-a--|-主键部分b-- (注意，这里不是ab, 而是只有b)

213

222

312

314

321

423

这个跟索引c的数据是一模一样的。

索引 cb 的组织是先按c排序，在按b排序，同时记录主键

-c--|-b--|-主键部分a-- (同上)

222

231

312

321

341

432

所以，结论是ca可以去掉，cb需要保留。

评论区留言点赞：

@浪里白条 帮大家总结了复习要点；

@约书亚 的问题里提到了MRR优化；

@HwangZHen 留言言简意赅。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



07 | 行锁功过：怎么减少行锁对性能的影响？

2018-11-28 林晓斌



在上一篇文章中，我跟你介绍了MySQL的全局锁和表级锁，今天我们就来讲讲MySQL的行锁。

MySQL的行锁是在引擎层由各个引擎自己实现的。但并不是所有的引擎都支持行锁，比如 MyISAM引擎就不支持行锁。不支持行锁意味着并发控制只能使用表锁，对于这种引擎的表，同一张表上任何时刻只能有一个更新在执行，这会影响到业务并发度。InnoDB是支持行锁的，这也是MyISAM被InnoDB替代的重要原因之一。

我们今天就主要来聊聊InnoDB的行锁，以及如何通过减少锁冲突来提升业务并发度。

顾名思义，行锁就是针对数据表中行记录的锁。这很好理解，比如事务A更新了一行，而这时候事务B也要更新同一行，则必须等事务A的操作完成后才能进行更新。

当然，数据库中还有一些没那么一目了然的概念和设计，这些概念如果理解和使用不当，容易导致程序出现非预期行为，比如两阶段锁。

从两阶段锁说起

我先给你举个例子。在下面的操作序列中，事务B的update语句执行时会是什么现象呢？假设字段id是表t的主键。

事务A	事务B
<pre>begin; update t set k=k+1 where id=1; update t set k=k+1 where id=2;</pre>	
	<pre>begin; update t set k=k+2 where id=1;</pre>
<pre>commit;</pre>	

这个问题的结论取决于事务A在执行完两条update语句后，持有哪些锁，以及在什么时候释放。你可以验证一下：实际上事务B的update语句会被阻塞，直到事务A执行commit之后，事务B才能继续执行。

知道了这个答案，你一定知道了事务A持有的两个记录的行锁，都是在commit的时候才释放的。

也就是说，在InnoDB事务中，行锁是在需要的时候才加上的，但并不是不需要了就立刻释放，而是要等到事务结束时才释放。这个就是两阶段锁协议。

知道了这个设定，对我们使用事务有什么帮助呢？那就是，**如果你的事务中需要锁多个行，要把最可能造成锁冲突、最可能影响并发度的锁尽量往后放**。我给你举个例子。

假设你负责实现一个电影票在线交易业务，顾客A要在影院B购买电影票。我们简化一点，这个业务需要涉及到以下操作：

1. 从顾客A账户余额中扣除电影票价；
2. 给影院B的账户余额增加这张电影票价；
3. 记录一条交易日志。

也就是说，要完成这个交易，我们需要update两条记录，并insert一条记录。当然，为了保证交

易的原子性，我们要把这三个操作放在一个事务中。那么，你会怎样安排这三个语句在事务中的顺序呢？

试想如果同时有另外一个顾客C要在影院B买票，那么这两个事务冲突的部分就是语句2了。因为它们要更新同一个影院账户的余额，需要修改同一行数据。

根据两阶段锁协议，不论你怎样安排语句顺序，所有的操作需要的行锁都是在事务提交的时候才释放的。所以，如果你把语句2安排在最后，比如按照3、1、2这样的顺序，那么影院账户余额这一行的锁时间就最少。这就最大程度地减少了事务之间的锁等待，提升了并发度。

好了，现在由于你的正确设计，影院余额这一行的行锁在一个事务中不会停留很长时间。但是，这并没有完全解决你的困扰。

如果这个影院做活动，可以低价预售一年内所有的电影票，而且这个活动只做一天。于是在活动时间开始的时候，你的MySQL就挂了。你登上服务器一看，CPU消耗接近100%，但整个数据库每秒就执行不到100个事务。这是什么原因呢？

这里，我就要说到死锁和死锁检测了。

死锁和死锁检测

当并发系统中不同线程出现循环资源依赖，涉及的线程都在等待别的线程释放资源时，就会导致这几个线程都进入无限等待的状态，称为死锁。这里我用数据库中的行锁举个例子。



这时候，事务A在等待事务B释放`id=2`的行锁，而事务B在等待事务A释放`id=1`的行锁。事务A和事务B在互相等待对方的资源释放，就是进入了死锁状态。当出现死锁以后，有两种策略：

- 一种策略是，直接进入等待，直到超时。这个超时时间可以通过参数`innodb_lock_wait_timeout`来设置。
- 另一种策略是，发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务，让其他事务得以继续执行。将参数`innodb_deadlock_detect`设置为`on`，表示开启这个逻辑。

在InnoDB中，`innodb_lock_wait_timeout`的默认值是50s，意味着如果采用第一个策略，当出现死锁以后，第一个被锁住的线程要过50s才会超时退出，然后其他线程才有可能继续执行。对于在线服务来说，这个等待时间往往是无法接受的。

但是，我们又不可能直接把这个时间设置成一个很小的值，比如1s。这样当出现死锁的时候，确实很快就可以解开，但如果不是死锁，而是简单的锁等待呢？所以，超时时间设置太短的话，会出现很多误伤。

所以，正常情况下我们还是要采用第二种策略，即：主动死锁检测，而且`innodb_deadlock_detect`的默认值本身就是`on`。主动死锁检测在发生死锁的时候，是能够快速发现并进行处理的，但是它也是有额外负担的。

你可以想象一下这个过程：每当一个事务被锁的时候，就要看看它所依赖的线程有没有被别人锁

住，如此循环，最后判断是否出现了循环等待，也就是死锁。

那如果是我们上面说到的所有事务都要更新同一行的场景呢？

每个新来的被堵住的线程，都要判断会不会由于自己的加入导致了死锁，这是一个时间复杂度是 $O(n)$ 的操作。假设有 1000 个并发线程要同时更新同一行，那么死锁检测操作就是 100 万这个量级的。虽然最终检测的结果是没有死锁，但是这期间要消耗大量的 CPU 资源。因此，你就会看到 CPU 利用率很高，但是每秒却执行不了几个事务。

根据上面的分析，我们来讨论一下，**怎么解决由这种热点行更新导致的性能问题呢？** 问题的症结在于，死锁检测要耗费大量的 CPU 资源。

一种头痛医头的方法，就是如果你能确保这个业务一定不会出现死锁，可以临时把死锁检测关掉。但是这种操作本身带有一定的风险，因为业务设计的时候一般不会把死锁当做一个严重错误，毕竟出现死锁了，就回滚，然后通过业务重试一般就没问题了，这是业务无损的。而关掉死锁检测意味着可能会出现大量的超时，这是业务有损的。

另一个思路是控制并发度。根据上面的分析，你会发现如果并发能够控制住，比如同一行同时最多只有 10 个线程在更新，那么死锁检测的成本很低，就不会出现这个问题。一个直接的想法就是，在客户端做并发控制。但是，你会很快发现这个方法不太可行，因为客户端很多。我见过一个应用，有 600 个客户端，这样即使每个客户端控制到只有 5 个并发线程，汇总到数据库服务端以后，峰值并发数也可能要达到 3000。

因此，这个并发控制要做在数据库服务端。如果你有中间件，可以考虑在中间件实现；如果你的团队有能修改 MySQL 源码的人，也可以做在 MySQL 里面。基本思路就是，对于相同行的更新，在进入引擎之前排队。这样在 InnoDB 内部就不会有大量的死锁检测工作了。

可能你会问，**如果团队里暂时没有数据库方面的专家，不能实现这样的方案，能不能从设计上优化这个问题呢？**

你可以考虑通过将一行改成逻辑上的多行来减少锁冲突。还是以影院账户为例，可以考虑放在多条记录上，比如 10 个记录，影院的账户总额等于这 10 个记录的值的总和。这样每次要给影院账户加金额的时候，随机选其中一条记录来加。这样每次冲突概率变成原来的 $1/10$ ，可以减少锁等待个数，也就减少了死锁检测的 CPU 消耗。

这个方案看上去是无损的，但其实这类方案需要根据业务逻辑做详细设计。如果账户余额可能会减少，比如退票逻辑，那么这时候就需要考虑当一部分行记录变成 0 的时候，代码要有特殊处理。

小结

今天，我和你介绍了 MySQL 的行锁，涉及了两阶段锁协议、死锁和死锁检测这两大部分内容。

其中，我以两阶段协议为起点，和你一起讨论了在开发的时候如何安排正确的事务语句。这里的原则/我给你的建议是：如果你的事务中需要锁多个行，要把最可能造成锁冲突、最可能影响并发度的锁的申请时机尽量往后放。

但是，调整语句顺序并不能完全避免死锁。所以我们引入了死锁和死锁检测的概念，以及提供了三个方案，来减少死锁对数据库的影响。减少死锁的主要方向，就是控制访问相同资源的并发事务量。

最后，我给你留下一个问题吧。如果你要删除一个表里面的前10000行数据，有以下三种方法可以做到：

- 第一种，直接执行**delete from T limit 10000;**
- 第二种，在一个连接中循环执行20次 **delete from T limit 500;**
- 第三种，在20个连接中同时执行**delete from T limit 500.**

你会选择哪一种方法呢？为什么呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期我给你留的问题是：当备库用-**single-transaction**做逻辑备份的时候，如果从主库的**binlog**传来一个**DDL**语句会怎么样？

假设这个**DDL**是针对表t1的，这里我把备份过程中几个关键的语句列出来：

```
Q1:SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Q2:START TRANSACTION WITH CONSISTENT SNAPSHOT;
/* other tables */
Q3:SAVEPOINT sp;
/* 时刻 1 */
Q4:show create table `t1`;
/* 时刻 2 */
Q5:SELECT * FROM `t1`;
/* 时刻 3 */
Q6:ROLLBACK TO SAVEPOINT sp;
/* 时刻 4 */
/* other tables */
```

在备份开始的时候，为了确保RR（可重复读）隔离级别，再设置一次RR隔离级别(Q1)；

启动事务，这里用 WITH CONSISTENT SNAPSHOT确保这个语句执行完就可以得到一个一致性视图 (Q2)；

设置一个保存点，这个很重要 (Q3)；

show create 是为了拿到表结构(Q4)，然后正式导数据 (Q5)，回滚到SAVEPOINT sp，在这里的作用是释放 t1的MDL锁 (Q6。当然这部分属于“超纲”，上文正文里面都没提到。

DDL从主库传过来的时间按照效果不同，我打了四个时刻。题目设定为小表，我们假定到达后，如果开始执行，则很快能够执行完成。

参考答案如下：

1. 如果在Q4语句执行之前到达，现象：没有影响，备份拿到的是DDL后的表结构。
2. 如果在“时刻 2”到达，则表结构被改过，Q5执行的时候，报 Table definition has changed, please retry transaction，现象：mysqldump终止；
3. 如果在“时刻2”和“时刻3”之间到达，mysqldump占着t1的MDL读锁，binlog被阻塞，现象：主从延迟，直到Q6执行完成。
4. 从“时刻4”开始，mysqldump释放了MDL读锁，现象：没有影响，备份拿到的是DDL前的表结构。

评论区留言点赞板：

@Aurora 给了最接近的答案；

@echo_陈 问了一个好问题；

@壹笙漂泊 做了很好的总结。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

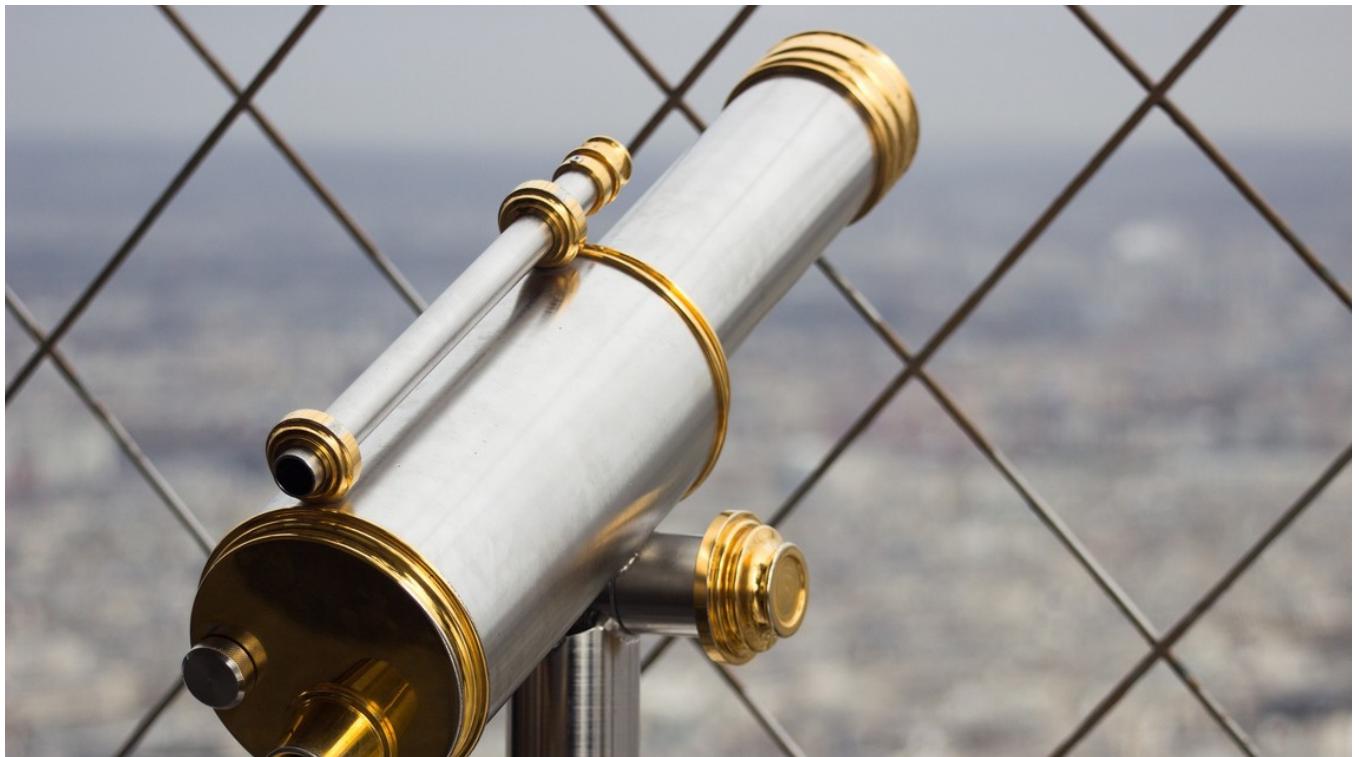
林晓斌

网名丁奇
前阿里资深技术专家



08 | 事务到底是隔离的还是不隔离的？

2018-11-30 林晓斌



我在第3篇文章和你讲事务隔离级别的时候提到过，如果是可重复读隔离级别，事务T启动的时候会创建一个视图**read-view**，之后事务T执行期间，即使有其他事务修改了数据，事务T看到的仍然跟在启动时看到的一样。也就是说，一个在可重复读隔离级别下执行的事务，好像与世无争，不受外界影响。

但是，我在上一篇文章中，和你分享行锁的时候又提到，一个事务要更新一行，如果刚好有另外一个事务拥有这一行的行锁，它又不能这么超然了，会被锁住，进入等待状态。问题是，既然进入了等待状态，那么等到这个事务自己获取到行锁要更新数据的时候，它读到的值又是什么呢？

我给你举一个例子吧。下面是一个只有两行的表的初始化语句。

```
mysql> CREATE TABLE `t` (
    `id` int(11) NOT NULL,
    `k` int(11) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;
insert into t(id, k) values(1,1),(2,2);
```

事务A	事务B	事务C
start transaction with consistent snapshot;		
	start transaction with consistent snapshot;	
		update t set k=k+1 where id=1;
	update t set k=k+1 where id=1; select k from t where id=1;	
select k from t where id=1; commit;		
	commit;	

图1 事务A、B、C的执行流程

这里，我们需要注意的是事务的启动时机。

begin/start transaction 命令并不是一个事务的起点，在执行到它们之后的第一个操作**InnoDB**表的语句，事务才真正启动。如果你想要马上启动一个事务，可以使用**start transaction with consistent snapshot** 这个命令。

还需要注意的是，在整个专栏里面，我们的例子中如果没有特别说明，都是默认 **autocommit=1**。

在这个例子中，事务**C**没有显式地使用**begin/commit**，表示这个**update**语句本身就是一个事务，语句完成的时候会自动提交。事务**B**在更新了行之后查询；事务**A**在一个只读事务中查询，并且时间顺序上是在事务**B**的查询之后。

这时，如果我告诉你事务**B**查到的k的值是3，而事务**A**查到的k的值是1，你是不是感觉有点晕呢？

所以，今天这篇文章，我其实就是想和你说明白这个问题，希望借由把这个疑惑解开的过程，能够帮助你对**InnoDB**的事务和锁有更进一步的理解。

在**MySQL**里，有两个“视图”的概念：

- 一个是**view**。它是一个用查询语句定义的虚拟表，在调用的时候执行查询语句并生成结果。创建视图的语法是**create view ...**，而它的查询方法与表一样。
- 另一个是**InnoDB**在实现**MVCC**时用到的一致性读视图，即**consistent read view**，用于支持

RC (Read Committed, 读提交) 和RR (Repeatable Read, 可重复读) 隔离级别的实现。

它没有物理结构，作用是事务执行期间用来定义“我能看到什么数据”。

在第3篇文章 [《事务隔离：为什么你改了我还看不见？》](#) 中，我跟你解释过一遍MVCC的实现逻辑。今天为了说明查询和更新的区别，我换一个方式来说明，把read view拆开。你可以结合这两篇文章的说明来更深一步地理解MVCC。

“快照”在MVCC里是怎么工作的？

在可重复读隔离级别下，事务在启动的时候就“拍了个快照”。注意，这个快照是基于整库的。

这时，你会说这看上去不太现实啊。如果一个库有100G，那么我启动一个事务，MySQL就要拷贝100G的数据出来，这个过程得多慢啊。可是，我平时的事务执行起来很快啊。

实际上，我们并不需要拷贝出这100G的数据。我们先来看看这个快照是怎么实现的。

InnoDB里面每个事务有一个唯一的事务ID，叫作transaction id。它是在事务开始的时候向InnoDB的事务系统申请的，是按申请顺序严格递增的。

而每行数据也都是有多个版本的。每次事务更新数据的时候，都会生成一个新的数据版本，并且把transaction id赋值给这个数据版本的事务ID，记为row trx_id。同时，旧的数据版本要保留，并且在新的数据版本中，能够有信息可以直接拿到它。

也就是说，数据表中的一行记录，其实可能有多个版本(row)，每个版本有自己的row trx_id。

如图2所示，就是一个记录被多个事务连续更新后的状态。

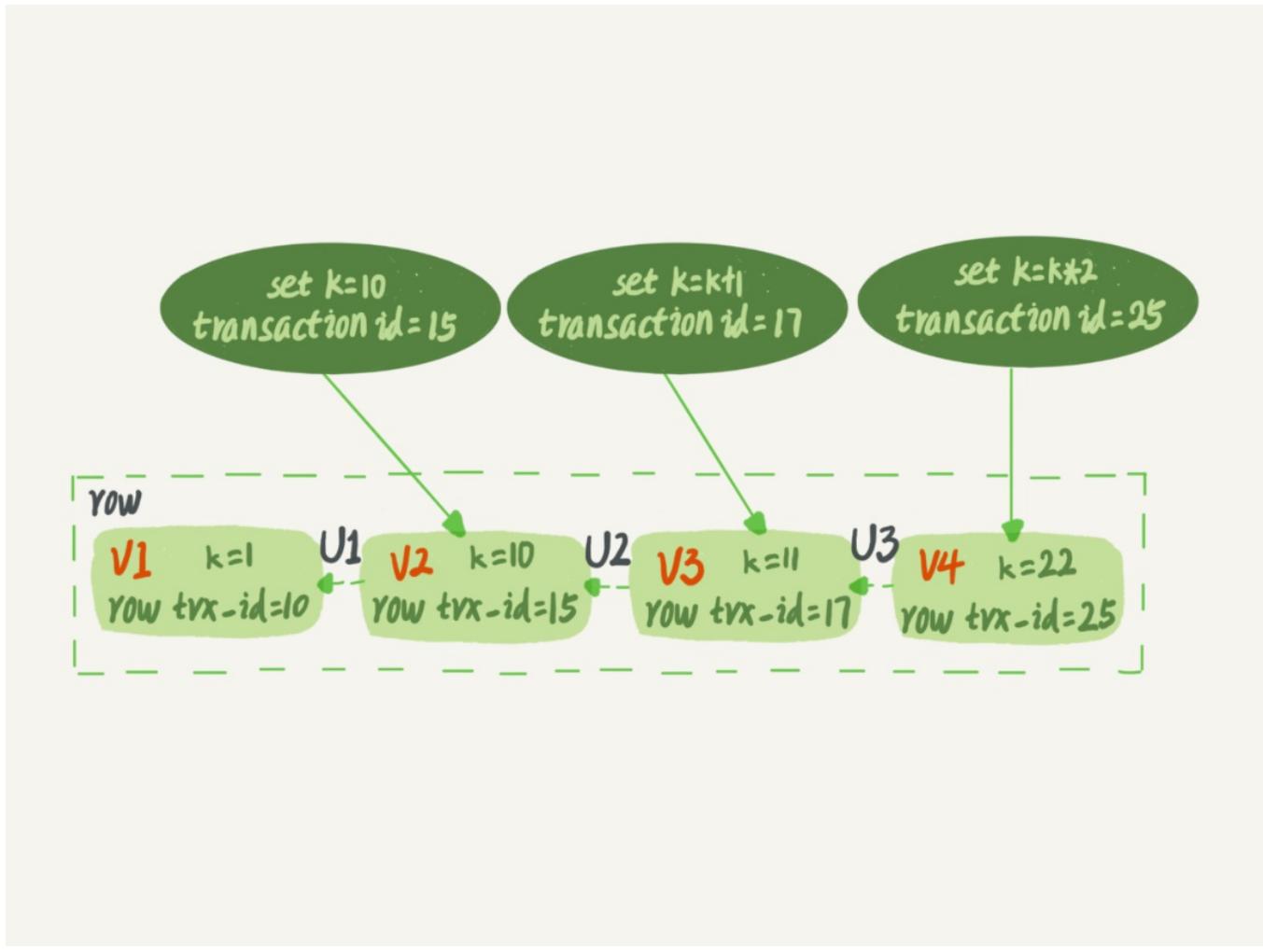


图2 行状态变更图

图中虚线框里是同一行数据的4个版本，当前最新版本是V4，k的值是22，它是被transaction id为25的事务更新的，因此它的row trx_id也是25。

你可能会问，前面的文章不是说，语句更新会生成undo log（回滚日志）吗？那么，**undo log在哪呢？**

实际上，图2中的三个虚线箭头，就是**undo log**；而V1、V2、V3并不是物理上真实存在的，而是每次需要的时候根据当前版本和**undo log**计算出来的。比如，需要V2的时候，就是通过V4依次执行U3、U2算出来。

明白了多版本和row trx_id的概念后，我们再来想一下，InnoDB是怎么定义那个“100G”的快照的。

按照可重复读的定义，一个事务启动的时候，能够看到所有已经提交的事务结果。但是之后，这个事务执行期间，其他事务的更新对它不可见。

因此，一个事务只需要在启动的时候声明说，“以我启动的时刻为准，如果一个数据版本是在我启动之前生成的，就认；如果是我启动以后才生成的，我就不认，我必须要找到它的上一个版本”。

当然，如果“上一个版本”也不可见，那就得继续往前找。还有，如果是这个事务自己更新的数据，它自己还是要认的。

在实现上，InnoDB为每个事务构造了一个数组，用来保存这个事务启动瞬间，当前正在“活跃”的所有事务ID。“活跃”指的就是，启动了但还没提交。

数组里面事务ID的最小值记为低水位，当前系统里面已经创建过的事务ID的最大值加1记为高水位。

这个视图数组和高水位，就组成了当前事务的一致性视图（read-view）。

而数据版本的可见性规则，就是基于数据的row trx_id和这个一致性视图的对比结果得到的。

这个视图数组把所有的row trx_id 分成了几种不同的情况。

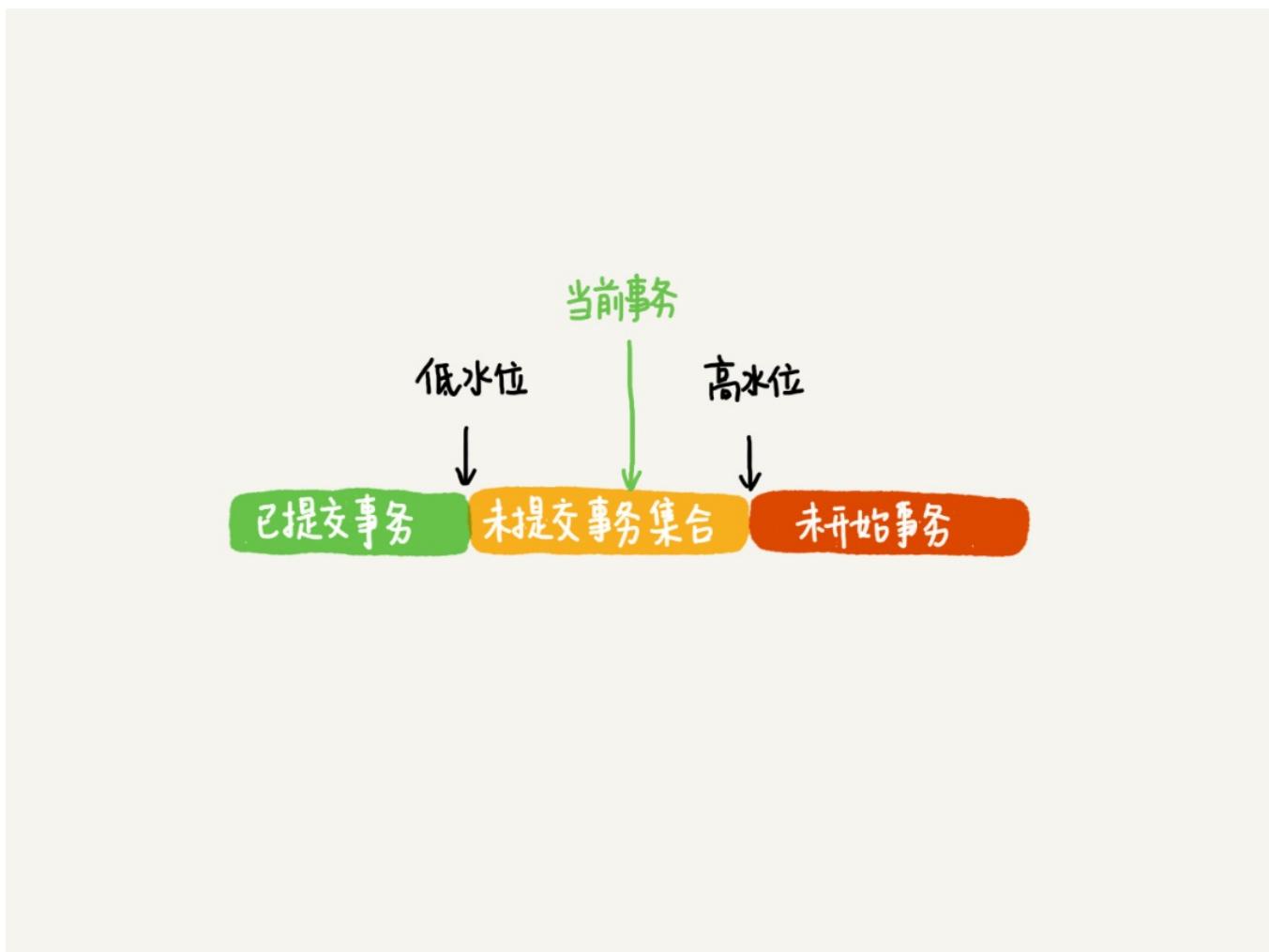


图3 数据版本可见性规则

这样，对于当前事务的启动瞬间来说，一个数据版本的row trx_id，有以下几种可能：

1. 如果落在绿色部分，表示这个版本是已提交的事务或者是当前事务自己生成的，这个数据是可见的；

2. 如果落在红色部分，表示这个版本是由将来启动的事务生成的，是肯定不可见的；
3. 如果落在黄色部分，那就包括两种情况
 - a. 若 `row trx_id` 在数组中，表示这个版本是由还没提交的事务生成的，不可见；
 - b. 若 `row trx_id` 不在数组中，表示这个版本是已经提交了的事务生成的，可见。

比如，对于图2中的数据来说，如果有一个事务，它的低水位是18，那么当它访问这一行数据时，就会从V4通过U3计算出V3，所以在它看来，这一行的值是11。

你看，有了这个声明后，系统里面随后发生的更新，是不是就跟这个事务看到的内容无关了呢？因为之后的更新，生成的版本一定属于上面的2或者3(a)的情况，而对它来说，这些新的数据版本是不存在的，所以这个事务的快照，就是“静态”的了。

所以你现在知道了，InnoDB利用了“所有数据都有多个版本”的这个特性，实现了“秒级创建快照”的能力。

接下来，我们继续看一下图1中的三个事务，分析下事务A的语句返回的结果，为什么是k=1。

这里，我们不妨做如下假设：

1. 事务A开始前，系统里面只有一个活跃事务ID是99；
2. 事务A、B、C的版本号分别是100、101、102，且当前系统里只有这四个事务；
3. 三个事务开始前，(1,1)这一行数据的`row trx_id`是90。

这样，事务A的视图数组就是[99,100]，事务B的视图数组是[99,100,101]，事务C的视图数组是[99,100,101,102]。

为了简化分析，我先把其他干扰语句去掉，只画出跟事务A查询逻辑有关的操作：

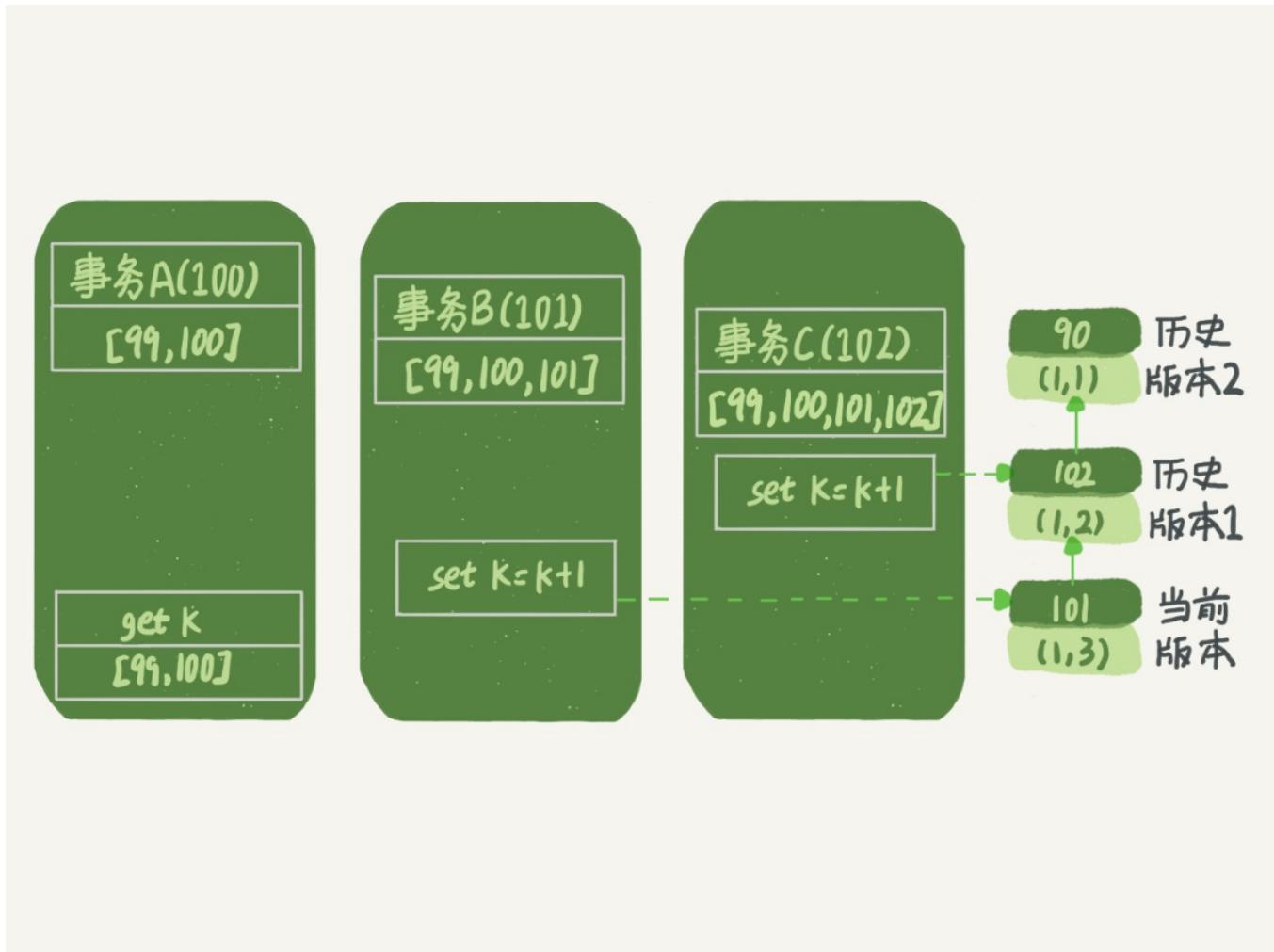


图4 事务A查询数据逻辑图

从图中可以看到，第一个有效更新是事务C，把数据从`(1,1)`改成了`(1,2)`。这时候，这个数据的最新版本的`row trx_id`是102，而90这个版本已经成为了历史版本。

第二个有效更新是事务B，把数据从`(1,2)`改成了`(1,3)`。这时候，这个数据的最新版本（即`row trx_id`）是101，而102又成为了历史版本。

你可能注意到了，在事务A查询的时候，其实事务B还没有提交，但是它生成的`(1,3)`这个版本已经变成当前版本了。但这个版本对事务A必须是不可见的，否则就变成脏读了。

好，现在事务A要来读数据了，它的视图数组是`[99,100]`。当然了，读数据都是从当前版本读起的。所以，事务A查询语句的读数据流程是这样的：

- 找到`(1,3)`的时候，判断出`row trx_id=101`，比高水位大，处于红色区域，不可见；
- 接着，找到上一个历史版本，一看`row trx_id=102`，比高水位大，处于红色区域，不可见；
- 再往前找，终于找到了`(1,1)`，它的`row trx_id=90`，比低水位小，处于绿色区域，可见。

这样执行下来，虽然期间这一行数据被修改过，但是事务A不论在什么时候查询，看到这行数据的结果都是一致的，所以我们称之为一致性读。

这个判断规则是从代码逻辑直接转译过来的，但是正如你所见，用于人肉分析可见性很麻烦。

所以，我来给你翻译一下。一个数据版本，对于一个事务视图来说，除了自己的更新总是可见以外，有三种情况：

1. 版本未提交，不可见；
2. 版本已提交，但是在视图创建后提交的，不可见；
3. 版本已提交，而且是在视图创建前提交的，可见。

现在，我们用这个规则来判断图4中的查询结果，事务A的查询语句的视图数组是在事务A启动的时候生成的，这时候：

- (1,3)还没提交，属于情况1，不可见；
- (1,2)虽然提交了，但是在视图数组创建之后提交的，属于情况2，不可见；
- (1,1)是在视图数组创建之前提交的，可见。

你看，去掉数字对比后，只用时间先后顺序来判断，分析起来是不是轻松多了。所以，后面我们就都用这个规则来分析。

更新逻辑

细心的同学可能有疑问了：事务B的**update**语句，如果按照一致性读，好像结果不对哦？

你看图5中，事务B的视图数组是先生成的，之后事务C才提交，不是应该看不见(1,2)吗，怎么能算出(1,3)来？

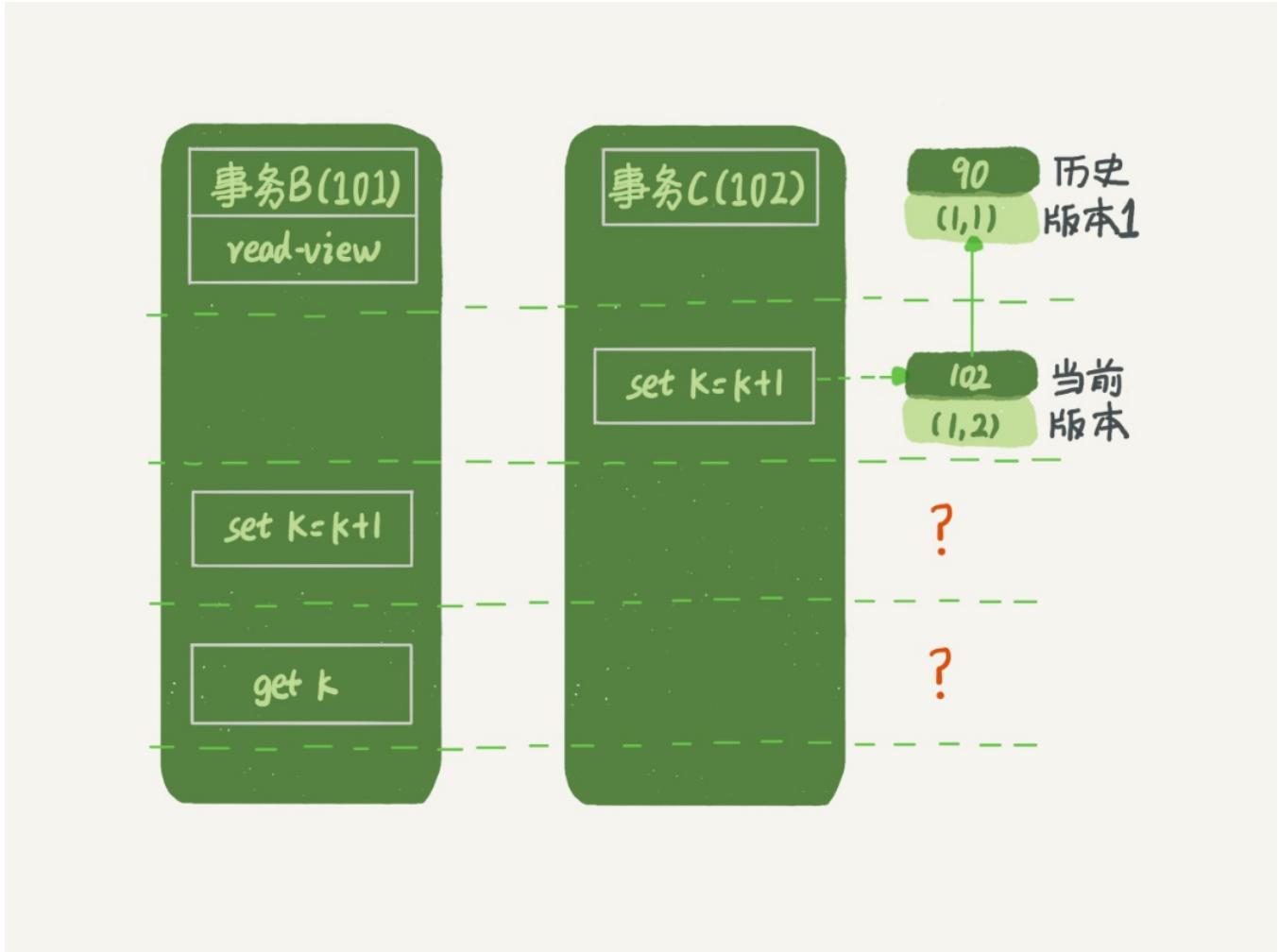


图5 事务B更新逻辑图

是的，如果事务B在更新之前查询一次数据，这个查询返回的k的值确实是1。

但是，当它要去更新数据的时候，就不能再在历史版本上更新了，否则事务C的更新就丢失了。因此，事务B此时的set k=k+1是在(1,2)的基础上进行的操作。

所以，这里就用到了这样一条规则：更新数据都是先读后写的，而这个读，只能读当前的值，称为“当前读”（current read）。

因此，在更新的时候，当前读拿到的数据是(1,2)，更新后生成了新版本的数据(1,3)，这个新版本的row trx_id是101。

所以，在执行事务B查询语句的时候，一看自己的版本号是101，最新数据的版本号也是101，是自己的更新，可以直接使用，所以查询得到的k的值是3。

这里我们提到了一个概念，叫作当前读。其实，除了update语句外，select语句如果加锁，也是当前读。

所以，如果把事务A的查询语句select * from t where id=1修改一下，加上lock in share mode 或 for update，也都可以读到版本号是101的数据，返回的k的值是3。下面这两个select语句，就是

分别加了读锁（S锁，共享锁）和写锁（X锁，排他锁）。

```
mysql> select k from t where id=1 lock in share mode;
mysql> select k from t where id=1 for update;
```

再往前一步，假设事务C不是马上提交的，而是变成了下面的事务C'，会怎么样呢？

事务A	事务B	事务C'
start transaction with consistent snapshot;		
	start transaction with consistent snapshot;	
		start transaction with consistent snapshot;
		update t set k=k+1 where id=1;
	update t set k=k+1 where id=1; select k from t where id=1;	
select k from t where id=1; commit;		commit;
	commit;	

图6 事务A、B、C'的执行流程

事务C'的不同是，更新后并没有马上提交，在它提交前，事务B的更新语句先发起了。前面说过了，虽然事务C还没提交，但是(1,2)这个版本也已经生成了，并且是当前的最新版本。那么，事务B的更新语句会怎么处理呢？

这时候，我们在上一篇文章中提到的“两阶段锁协议”就要上场了。事务C'没提交，也就是说(1,2)这个版本上的写锁还没释放。而事务B是当前读，必须要读最新版本，而且必须加锁，因此就被锁住了，必须等到事务C'释放这个锁，才能继续它的当前读。

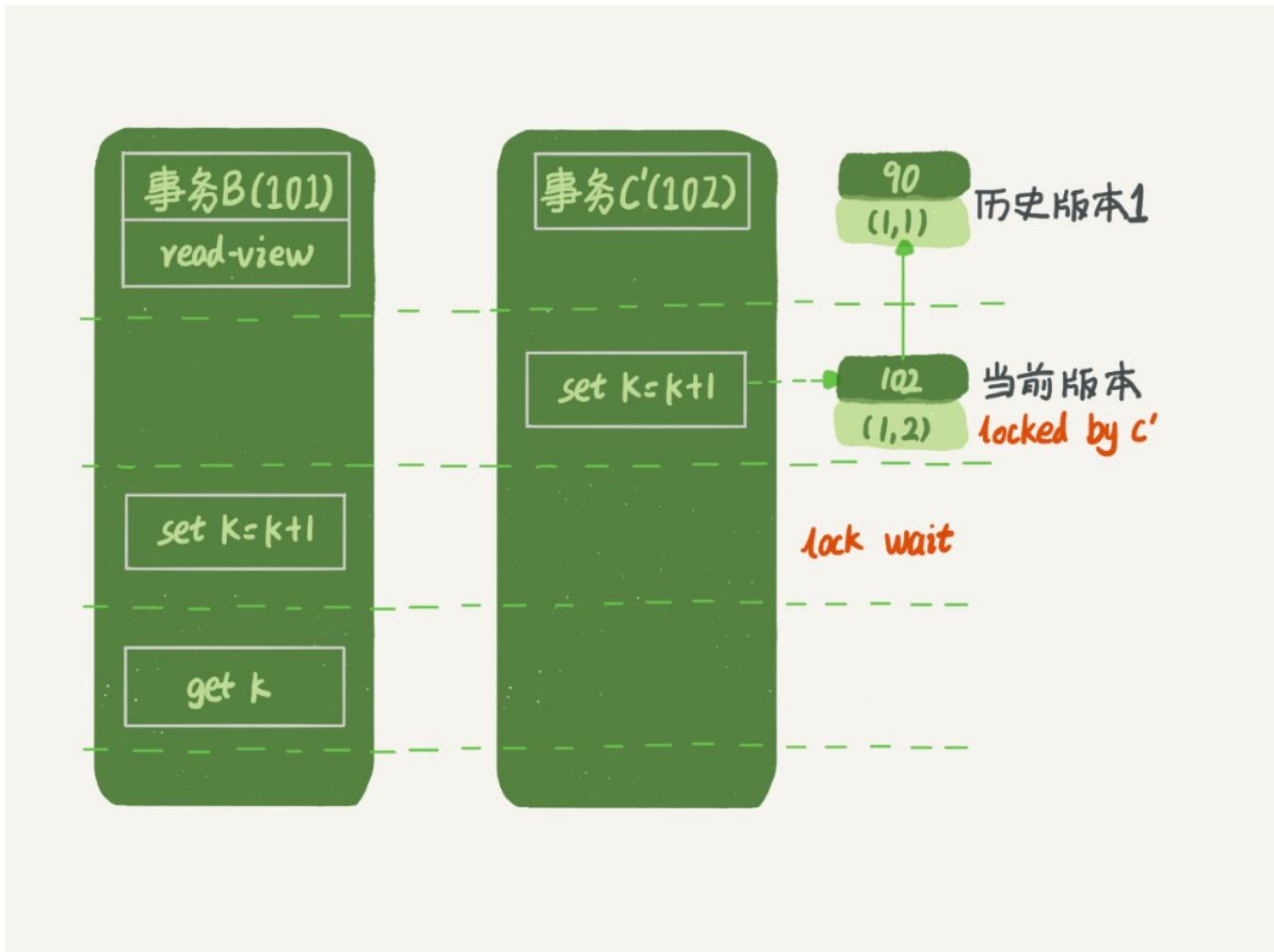


图7 事务B更新逻辑图（配合事务C）

到这里，我们把一致性读、当前读和行锁就串起来了。

现在，我们再回到文章开头的问题：事务的可重复读的能力是怎么实现的？

可重复读的核心就是一致性读（consistent read）；而事务更新数据的时候，只能用当前读。如果当前的记录的行锁被其他事务占用的话，就需要进入锁等待。

而读提交的逻辑和可重复读的逻辑类似，它们最主要的区别是：

- 在可重复读隔离级别下，只需要在事务开始的时候创建一致性视图，之后事务里的其他查询都共用这个一致性视图；
- 在读提交隔离级别下，每一个语句执行前都会重新算出一个新的视图。

那么，我们再看一下，在读提交隔离级别下，事务A和事务B的查询语句查到的k，分别应该是多少呢？

这里需要说明一下，“`start transaction with consistent snapshot;`”的意思是从这个语句开始，创建一个持续整个事务的一致性快照。所以，在读提交隔离级别下，这个用法就没意义了，等效于普通的`start transaction`。

下面是读提交时的状态图，可以看到这两个查询语句的创建视图数组的时机发生了变化，就是图中的**read view**框。（注意：这里，我们用的还是事务C的逻辑直接提交，而不是事务C）

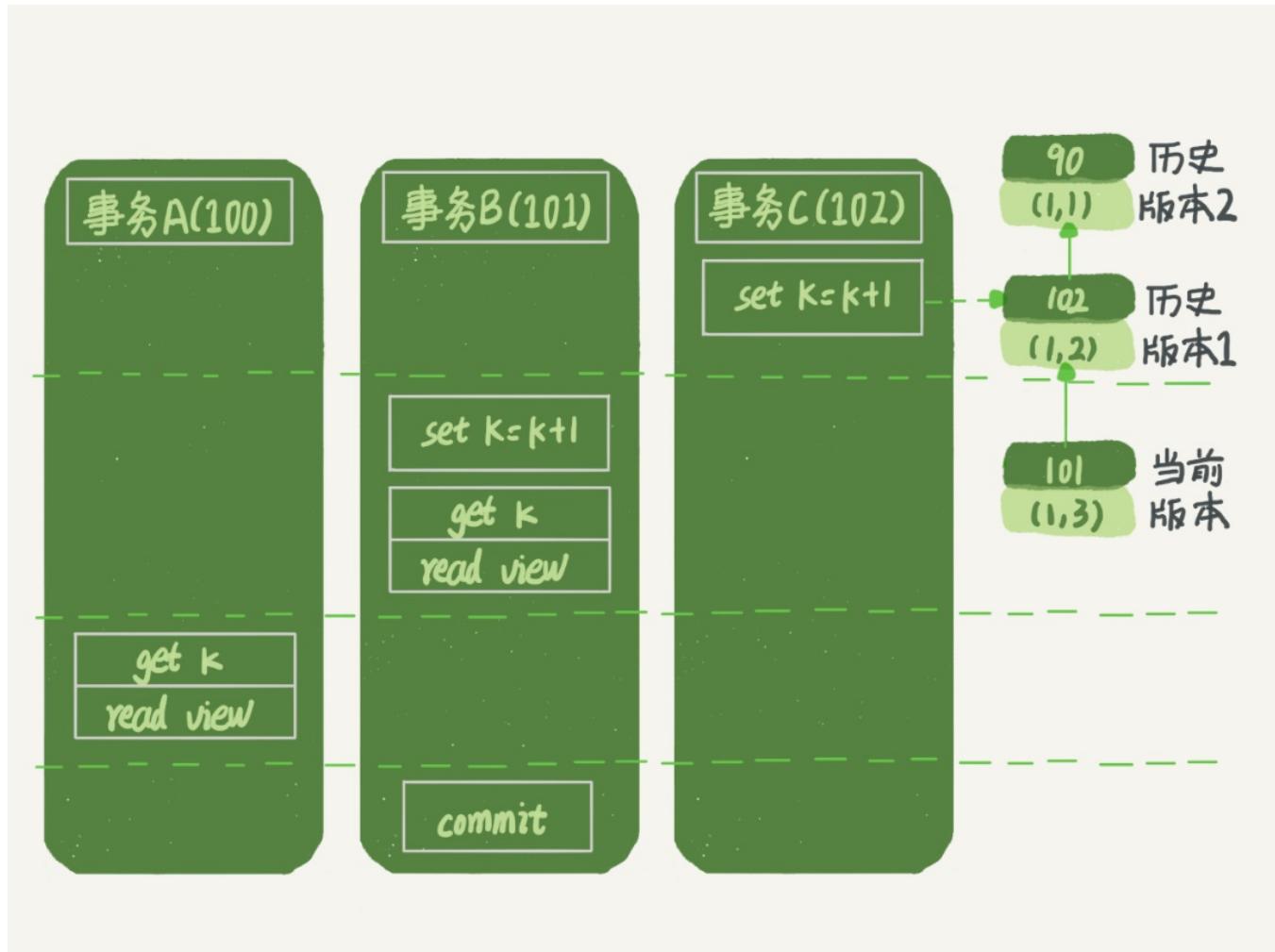


图8 读提交隔离级别下的事务状态图

这时，事务A的查询语句的视图数组是在执行这个语句的时候创建的，时序上(1,2)、(1,3)的生成时间都在创建这个视图数组的时刻之前。但是，在这个时刻：

- (1,3)还没提交，属于情况1，不可见；
- (1,2)提交了，属于情况3，可见。

所以，这时候事务A查询语句返回的是k=2。

显然地，事务B查询结果k=3。

小结

InnoDB的行数据有多个版本，每个数据版本有自己的row trx_id，每个事务或者语句都有自己的一致性视图。普通查询语句是一致性读，一致性读会根据row trx_id和一致性视图确定数据版本的可见性。

- 对于可重复读，查询只承认在事务启动前就已经提交完成的数据；

- 对于读提交，查询只承认在语句启动前就已经提交完成的数据；

而当前读，总是读取已经提交完成的最新版本。

你也可以想一下，为什么表结构不支持“可重复读”？这是因为表结构没有对应的行数据，也没有 `row trx_id`，因此只能遵循当前读的逻辑。

当然，MySQL 8.0 已经可以把表结构放在 InnoDB 字典里了，也许以后会支持表结构的可重复读。

又到思考题时间了。我用下面的表结构和初始化语句作为试验环境，事务隔离级别是可重复读。现在，我要把所有“字段 `c` 和 `id` 值相等的行”的 `c` 值清零，但是却发现了一个“诡异”的、改不掉的情况。请你构造出这种情况，并说明其原理。

```
mysql> CREATE TABLE `t` (
    `id` int(11) NOT NULL,
    `c` int(11) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;
insert into t(id, c) values(1,1),(2,2),(3,3),(4,4);
```

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t;
+---+---+
| id | c   |
+---+---+
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
| 4  | 4   |
+---+---+
4 rows in set (0.00 sec)

mysql> update t set c=0 where id=c;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> select * from t;
+---+---+
| id | c   |
+---+---+
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
| 4  | 4   |
+---+---+
4 rows in set (0.00 sec)
```

复现出来以后，请你再思考一下，在实际的业务开发中有没有可能碰到这种情况？你的应用代码会不会掉进这个“坑”里，你又是怎么解决的呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章最后，留给你的问题是：怎么删除表的前10000行。比较多的留言都选择了第二种方式，即：在一个连接中循环执行20次 `delete from T limit 500`。

确实是这样的，第二种方式是相对较好的。

第一种方式（即：直接执行`delete from T limit 10000`）里面，单个语句占用时间长，锁的时间也比较长；而且大事务还会导致主从延迟。

第三种方式（即：在20个连接中同时执行`delete from T limit 500`），会人为造成锁冲突。

评论区留言点赞板：

@Tony Du的评论，详细而且准确。

@Knight²⁰¹ 提到了如果可以加上特定条件，将这10000行天然分开，可以考虑第三种。是的，实际上在操作的时候我也建议你尽量拿到ID再删除。

@荒漠甘泉 提了一个不错的问题，大家需要区分行锁、MDL锁和表锁的区别。对InnoDB表更新一行，可能过了MDL关，却被挡在行锁阶段。

The image is a promotional graphic for a MySQL course. On the left, there's a logo for '极客时间' (Geek Time) featuring a stylized orange 'G' icon. The main title 'MySQL 实战 45 讲' is displayed in large, bold, dark gray font. Below it, a subtitle '从原理到实战，丁奇带你搞懂 MySQL' is shown in a smaller, regular black font. On the right side of the title, there's a portrait photo of a man with short dark hair and glasses, wearing a black long-sleeved shirt, with his arms crossed. At the bottom left, the teacher's name '林晓斌' is written in a large, dark font, followed by '网名丁奇' and '前阿里资深技术专家' in a smaller font.

09 | 普通索引和唯一索引，应该怎么选择？

2018-12-03 林晓斌



今天的正文开始前，我要特意感谢一下评论区几位留下高质量留言的同学。

用户名是 @某、人 的同学，对文章的知识点做了梳理，然后提了关于事务可见性的问题，就是先启动但是后提交的事务，对数据可见性的影响。@夏日雨同学也提到了这个问题，我在置顶评论中回复了，今天的文章末尾也会再展开说明。@Justin和@倪大人两位同学提了两个好问题。

对于能够引发更深一步思考的问题，我会在回复的内容中写上“好问题”三个字，方便你搜索，你也可以去看看他们的留言。

非常感谢大家很细致地看文章，并且留下了那么多和很高质量的留言。知道文章有给大家带来一些新理解，对我来说是一个很好的鼓励。同时，也让其他认真看评论区的同学，有机会发现一些自己还没有意识到的、但可能还不清晰的知识点，这也在总体上提高了整个专栏的质量。再次谢谢你们。

好了，现在就回到我们今天的正文内容。

在前面的基础篇文章中，我给你介绍过索引的基本概念，相信你已经了解了唯一索引和普通索引的区别。今天我们就继续来谈谈，在不同的业务场景下，应该选择普通索引，还是唯一索引？

假设你在维护一个市民系统，每个人都只有一个唯一的身份证号，而且业务代码已经保证了不会写入两个重复的身份证号。如果市民系统需要按照身份证号查姓名，就会执行类似这样的SQL语句：

```
select name from CUser where id_card = 'xxxxxxxxyyyyyyzzzz';
```

所以，你一定会考虑在**id_card**字段上建索引。

由于身份证号字段比较大，我不建议你把身份证号当做主键，那么现在你有两个选择，要么给**id_card**字段创建唯一索引，要么创建一个普通索引。如果业务代码已经保证了不会写入重复的身份证号，那么这两个选择逻辑上都是正确的。

现在我要问你的是，从性能的角度考虑，你选择唯一索引还是普通索引呢？选择的依据是什么呢？

简单起见，我们还是用第4篇文章[《深入浅出索引（上）》](#)中的例子来说明，假设字段 **k** 上的值都不重复。

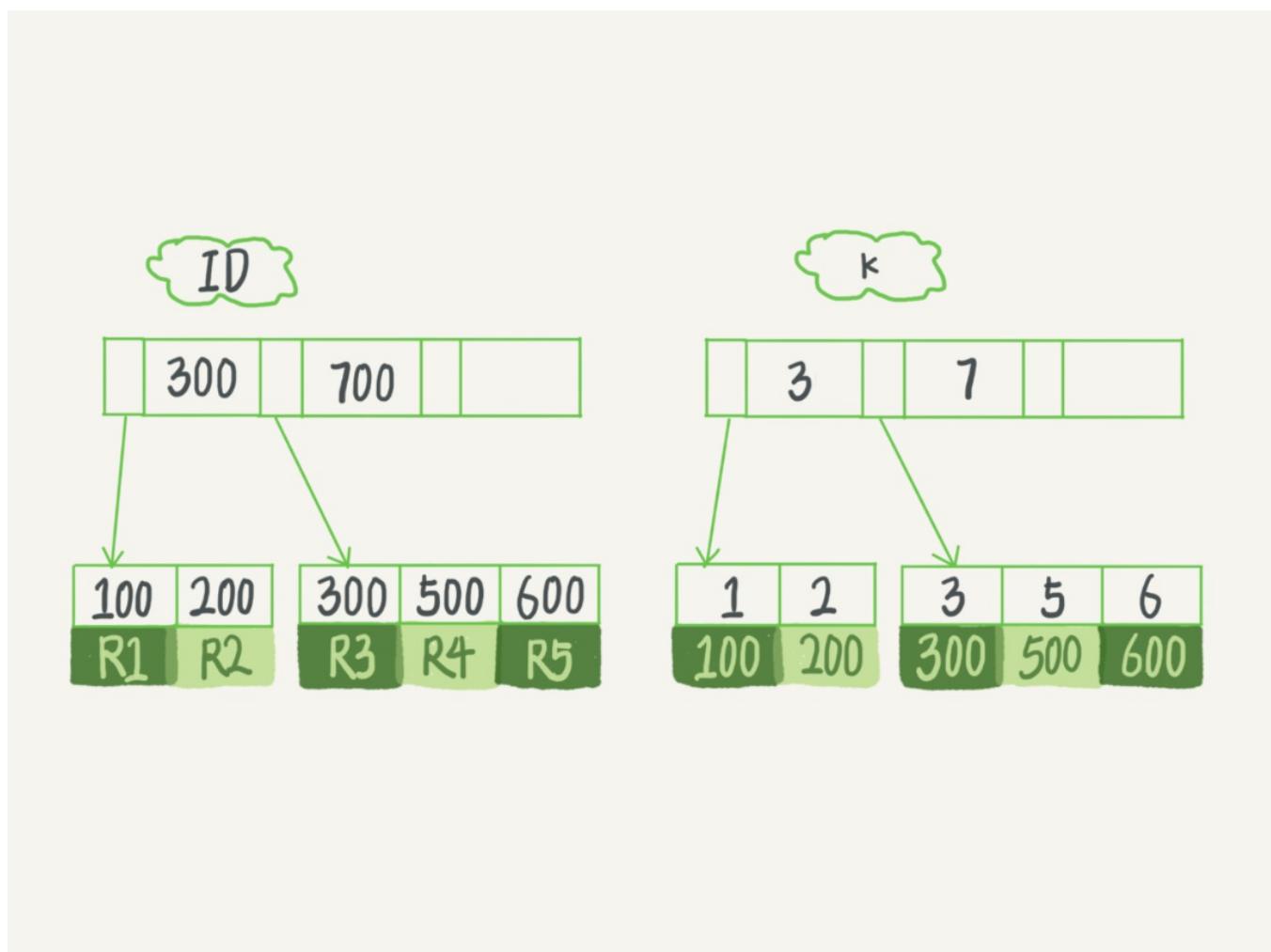


图1 InnoDB的索引组织结构

接下来，我们就从这两种索引对查询语句和更新语句的性能影响来进行分析。

查询过程

假设，执行查询的语句是 `select id from T where k=5`。这个查询语句在索引树上查找的过程，先是通过B+树从树根开始，按层搜索到叶子节点，也就是图中右下角的这个数据页，然后可以认为数据页内部通过二分法来定位记录。

- 对于普通索引来说，查找到满足条件的第一个记录(5,500)后，需要查找下一个记录，直到碰到第一个不满足k=5条件的记录。
- 对于唯一索引来说，由于索引定义了唯一性，查找到第一个满足条件的记录后，就会停止继续检索。

那么，这个不同带来的性能差距会有多少呢？答案是，微乎其微。

你知道的，InnoDB的数据是按数据页为单位来读写的。也就是说，当需要读一条记录的时候，并不是将这个记录本身从磁盘读出来，而是以页为单位，将其整体读入内存。在InnoDB中，每个数据页的大小默认是16KB。

因为引擎是按页读写的，所以说，当找到k=5的记录的时候，它所在的数据页就都在内存里了。那么，对于普通索引来说，要多做的那一次“查找和判断下一条记录”的操作，就只需要一次指针寻找和一次计算。

当然，如果k=5这个记录刚好是这个数据页的最后一个记录，那么要取下一个记录，必须读取下一个数据页，这个操作会稍微复杂一些。

但是，我们之前计算过，对于整型字段，一个数据页可以放近千个key，因此出现这种情况的概率会很低。所以，我们计算平均性能差异时，仍可以认为这个操作成本对于现在的CPU来说可以忽略不计。

更新过程

为了说明普通索引和唯一索引对更新语句性能的影响这个问题，我需要先跟你介绍一下change buffer。

当需要更新一个数据页时，如果数据页在内存中就直接更新，而如果这个数据页还没有在内存中的话，在不影响数据一致性的前提下，InnoDB会将这些更新操作缓存在change buffer中，这样就不需要从磁盘中读入这个数据页了。在下次查询需要访问这个数据页的时候，将数据页读入内存，然后执行change buffer中与这个页有关的操作。通过这种方式就能保证这个数据逻辑的正确性。

需要说明的是，虽然名字叫作change buffer，实际上它是可以持久化的数据。也就是说，change buffer在内存中有拷贝，也会被写入到磁盘上。

将change buffer中的操作应用到原数据页，得到最新结果的过程称为merge。除了访问这个数据页会触发merge外，系统有后台线程会定期merge。在数据库正常关闭(shutdown)的过程中，

也会执行merge操作。

显然，如果能够将更新操作先记录在**change buffer**，减少读磁盘，语句的执行速度会得到明显的提升。而且，数据读入内存是需要占用**buffer pool**的，所以这种方式还能够避免占用内存，提高内存利用率。

那么，什么条件下可以使用**change buffer**呢？

对于唯一索引来说，所有的更新操作都要先判断这个操作是否违反唯一性约束。比如，要插入(4,400)这个记录，就要先判断现在表中是否已经存在k=4的记录，而这必须要将数据页读入内存才能判断。如果都已经读入到内存了，那直接更新内存会更快，就没必要使用**change buffer**了。

因此，唯一索引的更新就不能使用**change buffer**，实际上也只有普通索引可以使用。

change buffer用的是**buffer pool**里的内存，因此不能无限增大。**change buffer**的大小，可以通过参数**innodb_change_buffer_max_size**来动态设置。这个参数设置为50的时候，表示**change buffer**的大小最多只能占用**buffer pool**的50%。

现在，你已经理解了**change buffer**的机制，那么我们再一起来看看如果要在这张表中插入一个新记录(4,400)的话，**InnoDB**的处理流程是怎样的。

第一种情况是，这个记录要更新的目标页在内存中。这时，**InnoDB**的处理流程如下：

- 对于唯一索引来说，找到3和5之间的位置，判断到没有冲突，插入这个值，语句执行结束；
- 对于普通索引来说，找到3和5之间的位置，插入这个值，语句执行结束。

这样看来，普通索引和唯一索引对更新语句性能影响的差别，只是一个判断，只会耗费微小的CPU时间。

但，这不是我们关注的重点。

第二种情况是，这个记录要更新的目标页不在内存中。这时，**InnoDB**的处理流程如下：

- 对于唯一索引来说，需要将数据页读入内存，判断到没有冲突，插入这个值，语句执行结束；
- 对于普通索引来说，则是将更新记录在**change buffer**，语句执行就结束了。

将数据从磁盘读入内存涉及随机IO的访问，是数据库里面成本最高的操作之一。**change buffer**因为减少了随机磁盘访问，所以对更新性能的提升是会很明显的。

之前我就碰到过一件事儿，有个**DBA**的同学跟我反馈说，他负责的某个业务的库内存命中率突然从99%降低到了75%，整个系统处于阻塞状态，更新语句全部堵住。而探究其原因后，我发现

这个业务有大量插入数据的操作，而他在前一天把其中的某个普通索引改成了唯一索引。

change buffer的使用场景

通过上面的分析，你已经清楚了使用**change buffer**对更新过程的加速作用，也清楚了**change buffer**只限于用在普通索引的场景下，而不适用于唯一索引。那么，现在有一个问题就是：普通索引的所有场景，使用**change buffer**都可以起到加速作用吗？

因为**merge**的时候是真正进行数据更新的时刻，而**change buffer**的主要目的就是将记录的变更动作缓存下来，所以在一个数据页做**merge**之前，**change buffer**记录的变更越多（也就是这个页面上要更新的次数越多），收益就越大。

因此，对于写多读少的业务来说，页面在写完以后马上被访问到的概率比较小，此时**change buffer**的使用效果最好。这种业务模型常见的就是账单类、日志类的系统。

反过来，假设一个业务的更新模式是写入之后马上会做查询，那么即使满足了条件，将更新先记录在**change buffer**，但之后由于马上要访问这个数据页，会立即触发**merge**过程。这样随机访问IO的次数不会减少，反而增加了**change buffer**的维护代价。所以，对于这种业务模式来说，**change buffer**反而起到了副作用。

索引选择和实践

回到我们文章开头的问题，普通索引和唯一索引应该怎么选择。其实，这两类索引在查询能力上是没差别的，主要考虑的是对更新性能的影响。所以，我建议你尽量选择普通索引。

如果所有的更新后面，都马上伴随着对这个记录的查询，那么你应该关闭**change buffer**。而在其他情况下，**change buffer**都能提升更新性能。

在实际使用中，你会发现，普通索引和**change buffer**的配合使用，对于数据量大的表的更新优化还是很明显的。

特别地，在使用机械硬盘时，**change buffer**这个机制的收效是非常显著的。所以，当你有一个类似“历史数据”的库，并且出于成本考虑用的是机械硬盘时，那你应该特别关注这些表里的索引，尽量使用普通索引，然后把**change buffer**尽量开大，以确保这个“历史数据”表的数据写入速度。

change buffer 和 redo log

理解了**change buffer**的原理，你可能会联想到我在前面文章中和你介绍过的**redo log**和**WAL**。

在前面文章的评论中，我发现有同学混淆了**redo log**和**change buffer**。**WAL** 提升性能的核心机制，也的确是尽量减少随机读写，这两个概念确实容易混淆。所以，这里我把它们放到了同一个流程里来说明，便于你区分这两个概念。

备注：这里，你可以再回顾下第2篇文章[《日志系统：一条SQL更新语句是如何执行的？》](#)中的相关内容。

现在，我们要在表上执行这个插入语句：

```
mysql> insert into t(id,k) values(id1,k1),(id2,k2);
```

这里，我们假设当前k索引树的状态，查找到位置后，**k1**所在的数据页在内存(**InnoDB buffer pool**)中，**k2**所在的数据页不在内存中。如图2所示是带**change buffer**的更新状态图。

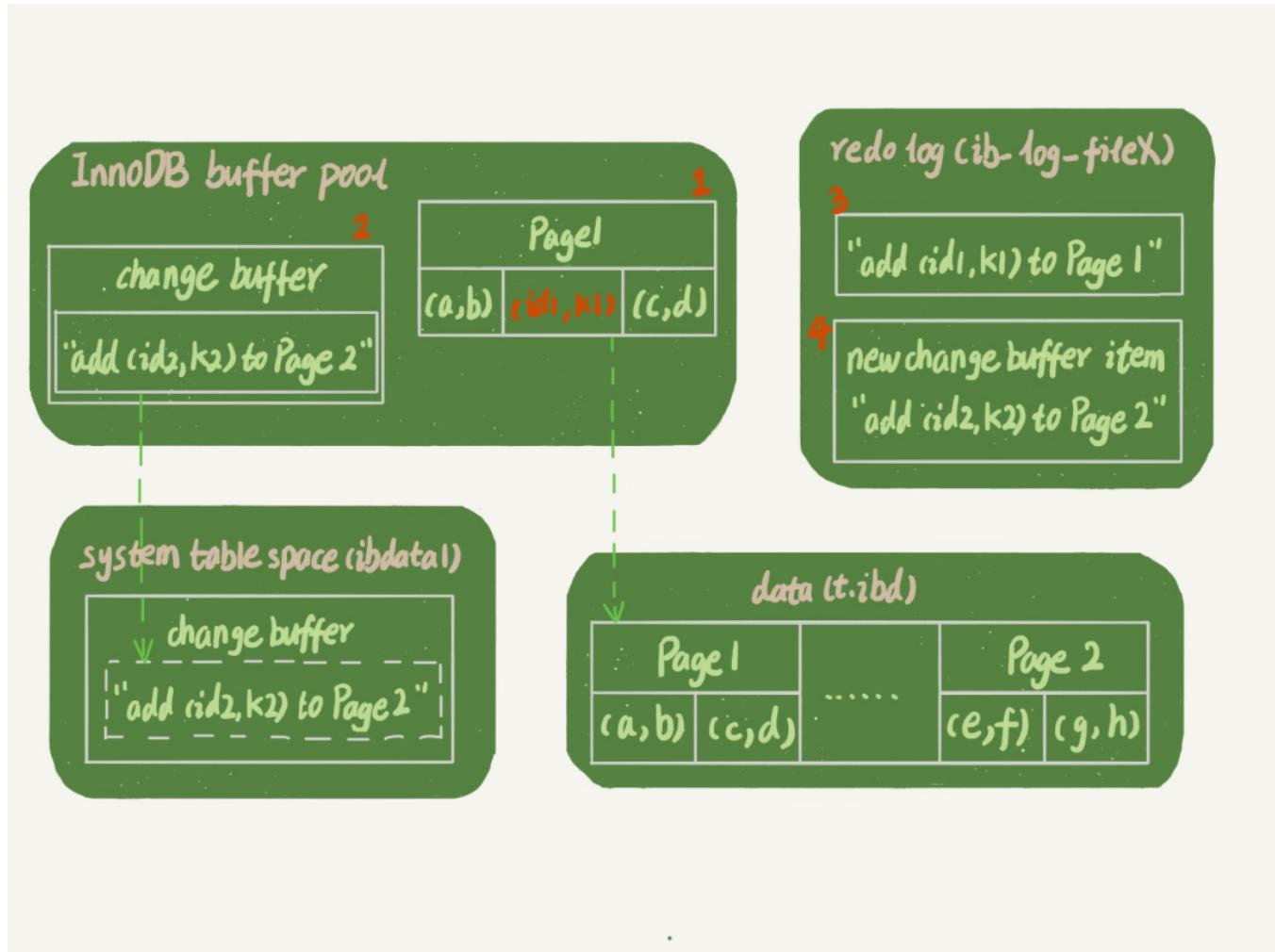


图2 带**change buffer**的更新过程

分析这条更新语句，你会发现它涉及了四个部分：内存、**redo log (ib_log_fileX)**、数据表空间(**t.ibd**)、系统表空间(**ibdata1**)。

这条更新语句做了如下的操作（按照图中的数字顺序）：

1. **Page 1**在内存中，直接更新内存；
2. **Page 2**没有在内存中，就在内存的**change buffer**区域，记录下“我要往**Page 2**插入一行”这个信息

3. 将上述两个动作记入redo log中（图中3和4）。

做完上面这些，事务就可以完成了。所以，你会看到，执行这条更新语句的成本很低，就是写了两处内存，然后写了一处磁盘（两次操作合在一起写了一次磁盘），而且还是顺序写的。

同时，图中的两个虚线箭头，是后台操作，不影响更新的响应时间。

那在这之后的读请求，要怎么处理呢？

比如，我们现在要执行 `select * from t where k in (k1, k2)`。这里，我画了这两个读请求的流程图。

如果读语句发生在更新语句后不久，内存中的数据都还在，那么此时的这两个读操作就与系统表空间（ibdata1）和 redo log（ib_log_fileX）无关了。所以，我在图中就没画出这两部分。

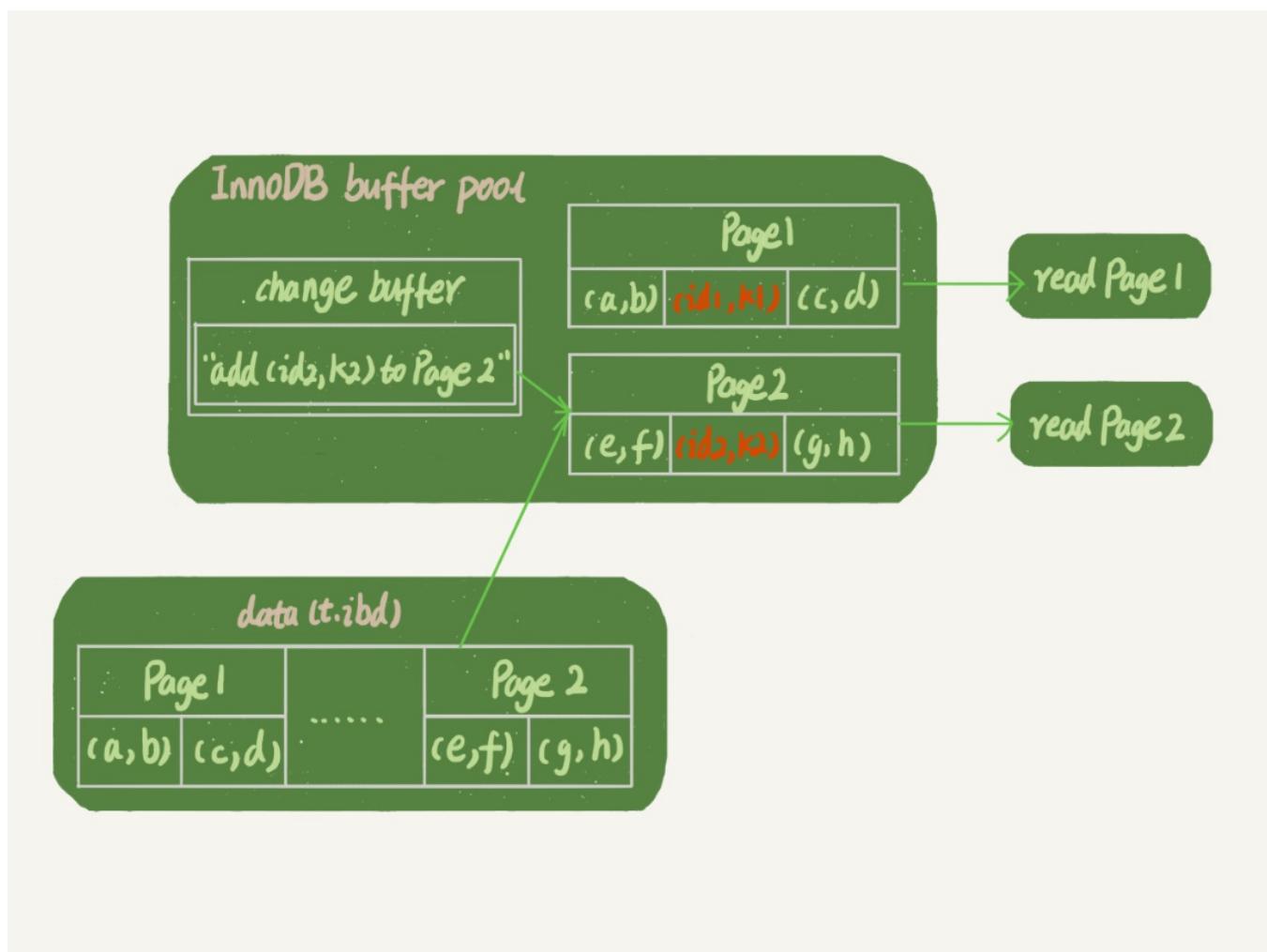


图3 带change buffer的读过程

从图中可以看到：

1. 读Page 1的时候，直接从内存返回。有几位同学在前面文章的评论中问到，WAL之后如果读数据，是不是一定要读盘，是不是一定要从redo log里面把数据更新以后才可以返回？其

实是不用的。你可以看一下图3的这个状态，虽然磁盘上还是之前的数据，但是这里直接从内存返回结果，结果是正确的。

- 要读Page 2的时候，需要把Page 2从磁盘读入内存中，然后应用**change buffer**里面的操作日志，生成一个正确的版本并返回结果。

可以看到，直到需要读Page 2的时候，这个数据页才会被读入内存。

所以，如果要简单地对比这两个机制在提升更新性能上的收益的话，**redo log** 主要节省的是随机写磁盘的IO消耗（转成顺序写），而**change buffer**主要节省的则是随机读磁盘的IO消耗。

小结

今天，我从普通索引和唯一索引的选择开始，和你分享了数据的查询和更新过程，然后说明了**change buffer**的机制以及应用场景，最后讲到了索引选择的实践。

由于唯一索引用不上**change buffer**的优化机制，因此如果业务可以接受，从性能角度出发我建议你优先考虑非唯一索引。

最后，又到了思考题时间。

通过图2你可以看到，**change buffer**一开始是写内存的，那么如果这个时候机器掉电重启，会不会导致**change buffer**丢失呢？**change buffer**丢失可不是小事，再从磁盘读入数据可就没有了**merge**过程，就等于是数据丢失了。会不会出现这种情况呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

补充：

评论区大家对“是否使用唯一索引”有比较多的讨论，主要是纠结在“业务可能无法确保”的情况。这里，我再说明一下：

- 首先，业务正确性优先。咱们这篇文章的前提是“业务代码已经保证不会写入重复数据”的情况下，讨论性能问题。如果业务不能保证，或者业务就是要求数据库来做约束，那么没得选，必须创建唯一索引。这种情况下，本篇文章的意义在于，如果碰上了大量插入数据慢、内存命中率低的时候，可以给你多提供一个排查思路。
- 然后，在一些“归档库”的场景，你是可以考虑使用唯一索引的。比如，线上数据只需要保留半年，然后历史数据保存在归档库。这时候，归档数据已经是确保没有唯一键冲突了。要提高归档效率，可以考虑把表里面的唯一索引改成普通索引。

上期问题时间

上期的问题是：如何构造一个“数据无法修改”的场景。评论区里已经有不少同学给出了正确答案，这里我再描述一下。

session A	session B
begin; select * from t;	update t set c=c+1;
update t set c=0 where id=c; select * from t;	

这样，**session A**看到的就是我截图的效果了。

其实，还有另外一种场景，同学们在留言区都还没有提到。

session A	session B'
	begin; select * from t;
begin; select * from t;	update t set c=c+1; commit;
update t set c=0 where id=c; select * from t;	

这个操作序列跑出来，**session A**看的内容也是能够复现我截图的效果的。这个**session B'**启动的事务比**A**要早，其实是上期我们描述事务版本的可见性规则时留的彩蛋，因为规则里还有一个“活跃事务的判断”，我是准备留到这里再补充的。

当我试图在这里讲述完整规则的时候，发现第8篇文章[《事务到底是隔离的还是不隔离的？》](#)中的解释引入了太多的概念，以至于分析起来非常复杂。

因此，我重写了第8篇，这样我们人工去判断可见性的时候，才会更方便。【看到这里，我建议你能够再重新打开第8篇文章并认真学习一次。如果学习的过程中，有任何问题，也欢迎你给我留言】

用新的方式来分析**session B'**的更新为什么对**session A**不可见就是：在**session A**视图数组创建的瞬间，**session B'**是活跃的，属于“版本未提交，不可见”这种情况。

业务中如果要绕过这类问题，@约书亚提供了一个“乐观锁”的解法，大家可以去上一篇的留言区看一下。

评论区留言点赞板：

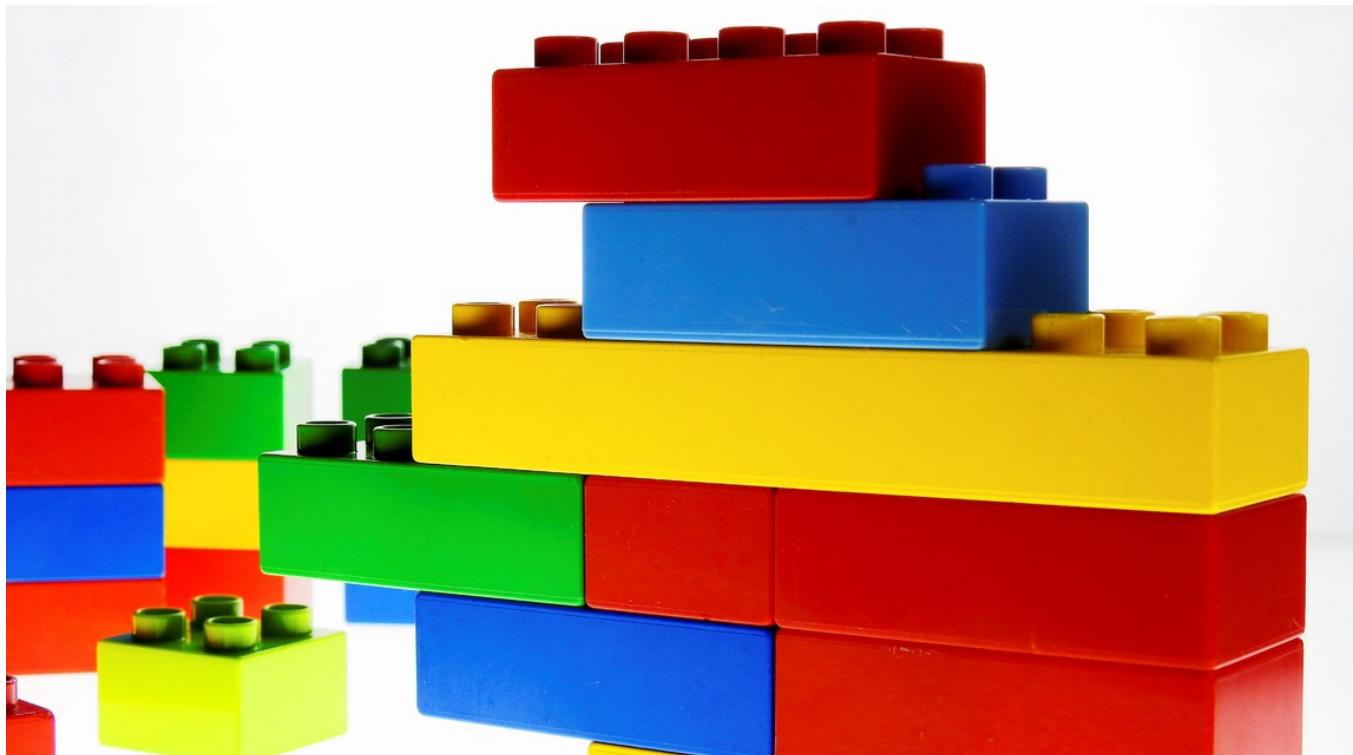
@某、人、@夏日雨、@周嶽、@李金刚 等同学提了一个很好的问题，就是我们今天答案的 session B' 的情况；

@justin 提到了提交和未提交版本的区别对待，@倪大人 提到了读提交和当前读的区别，都是经过了思考后提出的好问题，大家可以去留言区看看。



10 | MySQL为什么有时候会选错索引？

2018-12-05 林晓斌



前面我们介绍过索引，你已经知道了在MySQL中一张表其实是可以支持多个索引的。但是，你写SQL语句的时候，并没有主动指定使用哪个索引。也就是说，使用哪个索引是由MySQL来确定的。

不知道你有没有碰到过这种情况，一条本来可以执行得很快的语句，却由于MySQL选错了索引，而导致执行速度变得很慢？

我们一起来看一个例子吧。

我们先建一个简单的表，表里有a、b两个字段，并分别建上索引：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `a` (`a`),
  KEY `b` (`b`)
) ENGINE=InnoDB;
```

然后，我们往表t中插入10万行记录，取值按整数递增，即：(1,1,1), (2,2,2), (3,3,3)直到(100000,100000,100000)。

我是用存储过程来插入数据的，这里我贴出来方便你复现：

```
delimiter ;;
create procedure idata()
begin
    declare i int;
    set i=1;
    while(i<=100000)do
        insert into t values(i, i, i);
        set i=i+1;
    end while;
end;;
delimiter ;
call idata();
```

接下来，我们分析一条SQL语句：

```
mysql> select * from t where a between 10000 and 20000;
```

你一定会说，这个语句还用分析吗，很简单呀，**a**上有索引，肯定是要使用索引**a**的。

你说得没错，图1显示的就是使用**explain**命令看到的这条语句的执行情况。

```
mysql> explain select * from t where a between 10000 and 20000;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | t     | NULL       | range | a           | a   | 5      | NULL | 10001 | 100.00  | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图1 使用**explain**命令查看语句执行情况

从图1看上去，这条查询语句的执行也确实符合预期，**key**这个字段值是'**a**'，表示优化器选择了索引**a**。

不过别急，这个案例不会这么简单。在我们已经准备好的包含了10万行数据的表上，我们再做如下操作。

session A	session B
start transaction with consistent snapshot;	
	delete from t; call idata();
	explain select * from t where a between 10000 and 20000;
commit;	

图2 session A和session B的执行流程

这里，**session A**的操作你已经很熟悉了，它就是开启了一个事务。随后，**session B**把数据都删除后，又调用了**idata**这个存储过程，插入了10万行数据。

这时候，**session B**的查询语句**select * from t where a between 10000 and 20000**就不会再选择索引**a**了。我们可以通过慢查询日志（**slow log**）来查看一下具体的执行情况。

为了说明优化器选择的结果是否正确，我增加了一个对照，即：使用**force index(a)**来让优化器强制使用索引**a**（这部分内容，我还会在这篇文章的后半部分中提到）。

下面的三条**SQL**语句，就是这个实验过程。

```
set long_query_time=0;
select * from t where a between 10000 and 20000; /*Q1*/
select * from t force index(a) where a between 10000 and 20000; /*Q2*/
```

- 第一句，是将慢查询日志的阈值设置为0，表示这个线程接下来的语句都会被记录入慢查询日志中；
- 第二句，Q1是**session B**原来的查询；
- 第三句，Q2是加了**force index(a)**来和**session B**原来的查询语句执行情况对比。

如图3所示是这三条**SQL**语句执行完成后的慢查询日志。

```
# Time: 2018-12-03T10:26:35.711526Z
# User@Host: root[root] @ localhost [127.0.0.1]  Id:      4
# Query_time: 0.040877  Lock_time: 0.000151 Rows_sent: 10001  Rows_examined: 100000
SET timestamp=1543832795;
select * from t where a between 10000 and 20000;

# Time: 2018-12-03T10:26:11.028703Z
# User@Host: root[root] @ localhost [127.0.0.1]  Id:      4
# Query_time: 0.021076  Lock_time: 0.000163 Rows_sent: 10001  Rows_examined: 10001
SET timestamp=1543832771;
select * from t force index(a) where a between 10000 and 20000;
```

图3 slow log结果

可以看到，Q1扫描了10万行，显然是走了全表扫描，执行时间是40毫秒。Q2扫描了10001行，执行了21毫秒。也就是说，我们在没有使用**force index**的时候，MySQL用错了索引，导致了更长的执行时间。

这个例子对应的是我们平常不断地删除历史数据和新增数据的场景。这时，MySQL竟然会选错索引，是不是有点奇怪呢？今天，我们就从这个奇怪的结果说起吧。

优化器的逻辑

在第一篇文章中，我们就提到过，选择索引是优化器的工作。

而优化器选择索引的目的，是找到一个最优的执行方案，并用最小的代价去执行语句。在数据库里面，扫描行数是影响执行代价的因素之一。扫描的行数越少，意味着访问磁盘数据的次数越少，消耗的CPU资源越少。

当然，扫描行数并不是唯一的判断标准，优化器还会结合是否使用临时表、是否排序等因素进行综合判断。

我们这个简单的查询语句并没有涉及到临时表和排序，所以MySQL选错索引肯定是在判断扫描行数的时候出问题了。

那么，问题就是：**扫描行数是怎么判断的？**

MySQL在真正开始执行语句之前，并不能精确地知道满足这个条件的记录有多少条，而只能根据统计信息来估算记录数。

这个统计信息就是索引的“区分度”。显然，一个索引上不同的值越多，这个索引的区分度就越好。而一个索引上不同的值的个数，我们称之为“基数”（**cardinality**）。也就是说，这个基数越大，索引的区分度越好。

我们可以使用**show index**方法，看到一个索引的基数。如图4所示，就是表t的**show index**的结果。虽然这个表的每一行的三个字段值都是一样的，但是在统计信息中，这三个索引的基数值并不相同，而且其实都不准确。

图4 表t的show index 结果

那么，MySQL是怎样得到索引的基数的呢？这里，我给你简单介绍一下MySQL采样统计的方法。

为什么要采样统计呢？因为把整张表取出来一行行统计，虽然可以得到精确的结果，但是代价太高了，所以只能选择“采样统计”。

采样统计的时候，InnoDB默认会选择N个数据页，统计这些页面上的不同值，得到一个平均值，然后乘以这个索引的页面数，就得到了这个索引的基数。

而数据表是会持续更新的，索引统计信息也不会固定不变。所以，当变更的数据行数超过 $1/M$ 的时候，会自动触发重新做一次索引统计。

在MySQL中，有两种存储索引统计的方式，可以通过设置参数`innodb_stats_persistent`的值来选择：

- 设置为on的时候，表示统计信息会持久化存储。这时，默认的N是20，M是10。
 - 设置为off的时候，表示统计信息只存储在内存中。这时，默认的N是8，M是16。

由于是采样统计，所以不管N是20还是8，这个基数都是很容易不准的。

但，这还不是全部。

你可以从图4中看到，这次的索引统计值（**cardinality**列）虽然不够精确，但大体上还是差不多的，选错索引一定还有别的原因。

其实索引统计只是一个输入，对于一个具体的语句来说，优化器还要判断，执行这个语句本身要扫描多少行。

接下来，我们再一起看看优化器预估的，这两个语句的扫描行数是多少。

图5 意外的explain结果

rows这个字段表示的是预计扫描行数。

其中，Q1的结果还是符合预期的，**rows**的值是104620；但是Q2的**rows**值是37116，偏差就大了。而图1中我们用**explain**命令看到的**rows**是只有10001行，是这个偏差误导了优化器的判断。

到这里，可能你的第一个疑问不是为什么不准，而是优化器为什么放着扫描37000行的执行计划不用，却选择了扫描行数是100000的执行计划呢？

这是因为，如果使用索引a，每次从索引a上拿到一个值，都要回到主键索引上查出整行数据，这个代价优化器也要算进去的。

而如果选择扫描10万行，是直接在主键索引上扫描的，没有额外的代价。

优化器会估算这两个选择的代价，从结果看来，优化器认为直接扫描主键索引更快。当然，从执行时间看来，这个选择并不是最优的。

使用普通索引需要把回表的代价算进去，在图1执行**explain**的时候，也考虑了这个策略的代价，但图1的选择是对的。也就是说，这个策略并没有问题。

所以冤有头债有主，MySQL选错索引，这件事儿还得归咎到没能准确地判断出扫描行数。至于为什么会得到错误的扫描行数，这个原因就作为课后问题，留给你去分析了。

既然是统计信息不对，那就修正。**analyze table t**命令，可以用来重新统计索引信息。我们来看一下执行效果。

```
mysql> analyze table t;
+-----+-----+-----+
| Table | Op   | Msg_type | Msg_text |
+-----+-----+-----+
| test.t | analyze | status   | OK      |
+-----+-----+-----+
1 row in set (0.01 sec)

mysql> explain select * from t where a between 10000 and 20000;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | partitions | type  | possible_keys | key   | key_len | ref    | rows   | filtered | Extra          |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE      | t     | NULL       | range | a           | a     | 5       | NULL    | 10001  | 100.00    | Using index condition |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

图6 执行**analyze table t**命令恢复的**explain**结果

这回对了。

所以在实践中，如果你发现**explain**的结果预估的**rows**值跟实际情况差距比较大，可以采用这个方法来处理。

其实，如果只是索引统计不准确，通过**analyze**命令可以解决很多问题，但是前面我们说了，优化器可不止是看扫描行数。

依然是基于这个表t，我们看看另外一个语句：

```
mysql> select * from t where (a between 1 and 1000) and (b between 50000 and 100000) order by
```

从条件上看，这个查询没有符合条件的记录，因此会返回空集合。

在开始执行这条语句之前，你可以先设想一下，如果你来选择索引，会选择哪一个呢？

为了便于分析，我们先来看一下a、b这两个索引的结构图。

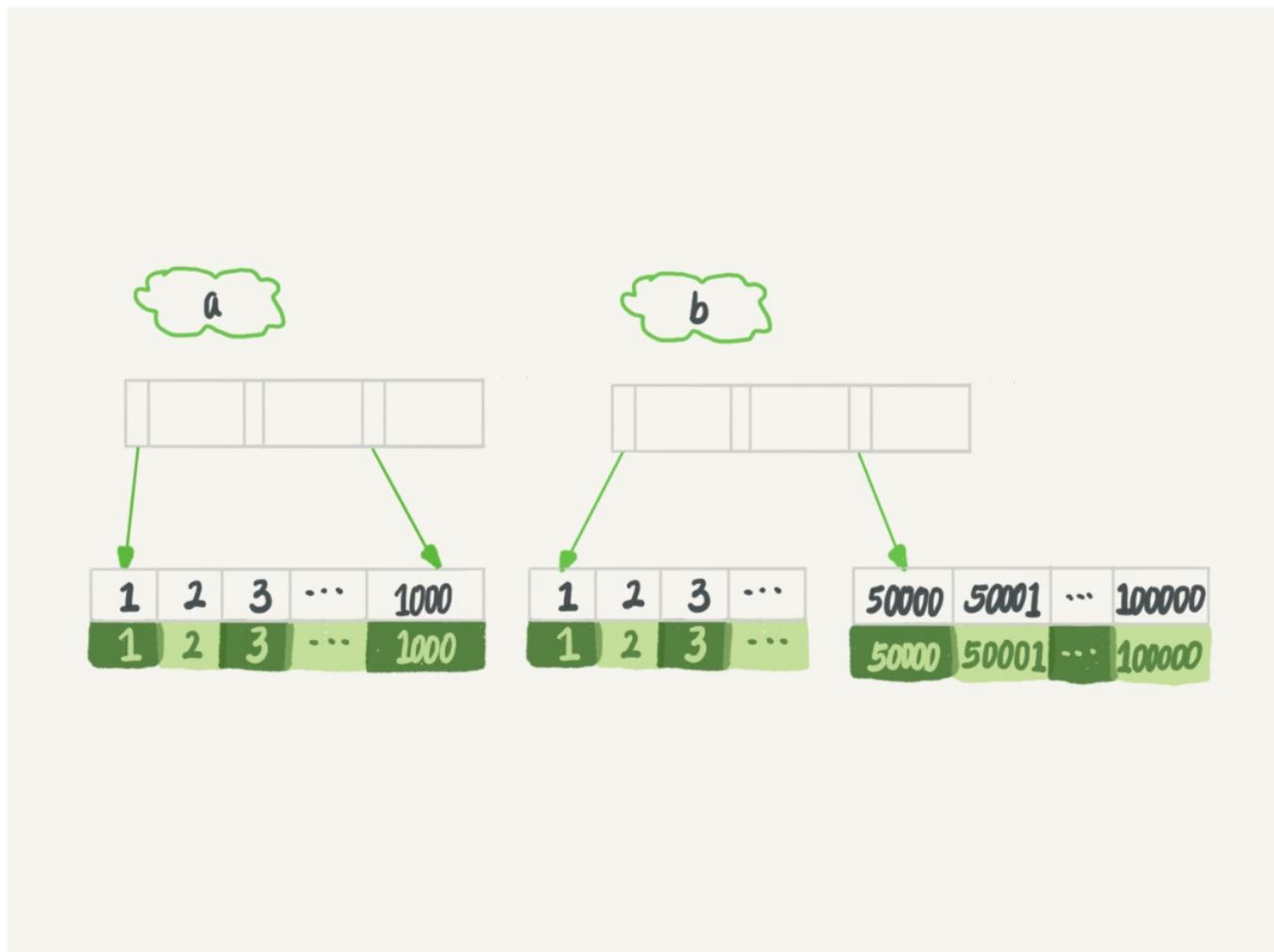


图7 a、b索引的结构图

如果使用索引**a**进行查询，那么就是扫描索引**a**的前1000个值，然后取到对应的**id**，再到主键索引上去查出每一行，然后根据字段**b**来过滤。显然这样需要扫描1000行。

如果使用索引**b**进行查询，那么就是扫描索引**b**的最后50001个值，与上面的执行过程相同，也是需要回到主键索引上取值再判断，所以需要扫描50001行。

所以你一定会想，如果使用索引**a**的话，执行速度明显会快很多。那么，下面我们就来看看到底是不是这么一回事儿。

图8是执行explain的结果。



```
mysql> explain select * from t where (a between 1 and 1000) and (b between 50000 and 100000) order by b limit 1;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t | NULL | range | a,b | b | 5 | NULL | 50198 |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图8 使用explain方法查看执行计划 2

可以看到，返回结果中**key**字段显示，这次优化器选择了索引**b**，而**rows**字段显示需要扫描的行数是**50198**。

从这个结果中，你可以得到两个结论：

1. 扫描行数的估计值依然不准确；
2. 这个例子里MySQL又选错了索引。

索引选择异常和处理

其实大多数时候优化器都能找到正确的索引，但偶尔你还是会碰到我们上面举例的这两种情况：原本可以执行得很快的SQL语句，执行速度却比你预期的慢很多，你应该怎么办呢？

一种方法是，像我们第一个例子一样，采用**force index**强行选择一个索引。MySQL会根据词法解析的结果分析出可能可以使用的索引作为候选项，然后在候选列表中依次判断每个索引需要扫描多少行。如果**force index**指定的索引在候选索引列表中，就直接选择这个索引，不再评估其他索引的执行代价。

我们来看看第二个例子。刚开始分析时，我们认为选择索引**a**会更好。现在，我们就来看看执行效果：

```
mysql> select * from t where a between 1 and 1000 and b between 50000 and 100000 order by b limit 1;
Empty set (2.23 sec)

mysql> select * from t force index(a) where a between 1 and 1000 and b between 50000 and 100000 order by b limit 1;
Empty set (0.05 sec)
```

图9 使用不同索引的语句执行耗时

可以看到，原本语句需要执行**2.23**秒，而当你使用**force index(a)**的时候，只用了**0.05**秒，比优化器的选择快了**40**多倍。

也就是说，优化器没有选择正确的索引，**force index**起到了“矫正”的作用。

不过很多程序员不喜欢使用**force index**，一来这么写不优美，二来如果索引改了名字，这个语句

也得改，显得很麻烦。而且如果以后迁移到别的数据库的话，这个语法还可能会不兼容。

但其实使用**force index**最主要的问题还是变更的及时性。因为选错索引的情况还是比较少出现的，所以开发的时候通常不会先写上**force index**。而是等到线上出现问题的时候，你才会再去修改SQL语句、加上**force index**。但是修改之后还要测试和发布，对于生产系统来说，这个过程不够敏捷。

所以，数据库的问题最好还是在数据库内部来解决。那么，在数据库里面该怎样解决呢？

既然优化器放弃了使用索引a，说明a还不够合适，所以第二种方法就是，我们可以考虑修改语句，引导MySQL使用我们期望的索引。比如，在这个例子里，显然把“order by b limit 1”改成“order by b,a limit 1”，语义的逻辑是相同的。

我们来看看改之后的效果：

图10 order by b,a limit 1 执行结果

之前优化器选择使用索引**b**, 是因为它认为使用索引**b**可以避免排序 (**b**本身是索引, 已经是有序的, 如果选择索引**b**的话, 不需要再做排序, 只需要遍历), 所以即使扫描行数多, 也判定为代价更小。

现在**order by b,a**这种写法，要求按照**b,a**排序，就意味着使用这两个索引都需要排序。因此，扫描行数成了影响决策的主要条件，于是此时优化器选了只需要扫描1000行的索引**a**。

当然，这种修改并不是通用的优化手段，只是刚好在这个语句里面有**limit 1**，因此如果有满足条件的记录，**order by b limit 1**和**order by b,a limit 1**都会返回**b**是最小的那一行，逻辑上一致，才可以这么做。

如果你觉得修改语义这件事儿不太好，这里还有一种改法，图11是执行效果。

```
mysql> select * from  (select * from t where (a between 1 and 1000)  and (b between 50000 and 100000) order by b limit 100)alias limit 1;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | <derived2> | NULL | ALL | NULL | NULL | NULL | NULL | 100 | 100.00 | NULL |
| 2 | DERIVED | t | NULL | range | a,b | a | 5 | NULL | 1000 | 50.00 | Using index condition; Using where; Using filesort |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图11 改写SQL的explain

在这个例子里，我们用**limit 100**让优化器意识到，使用**b**索引代价是很高的。其实是我们根据数

据特征诱导了一下优化器，也不具备通用性。

第三种方法是，在有些场景下，我们可以新建一个更合适的索引，来提供给优化器做选择，或删掉误用的索引。

不过，在这个例子中，我没有找到通过新增索引来改变优化器行为的方法。这种情况其实比较少，尤其是经过**DBA**索引优化过的库，再碰到这个**bug**，找到一个更合适的索引一般比较难。

如果说还有一个方法是删掉索引**b**，你可能会觉得好笑。但实际上我碰到过两次这样的例子，最终是**DBA**跟业务开发沟通后，发现这个优化器错误选择的索引其实根本没有必要存在，于是就删掉了这个索引，优化器也就重新选择到了正确的索引。

小结

今天我们一起聊了聊索引统计的更新机制，并提到了优化器存在选错索引的可能性。

对于由于索引统计信息不准确导致的问题，你可以用**analyze table**来解决。

而对于其他优化器误判的情况，你可以在应用端用**force index**来强行指定索引，也可以通过修改语句来引导优化器，还可以通过增加或者删除索引来绕过这个问题。

你可能会说，今天这篇文章后面的几个例子，怎么都没有展开说明其原理。我要告诉你的是，今天的话题，我们面对的是**MySQL**的**bug**，每一个展开都必须深入到一行行代码去量化，实在不是我们在这里应该做的事情。

所以，我把用过的解决方法跟你分享，希望你在碰到类似情况的时候，能够有一些思路。

你平时在处理**MySQL**优化器**bug**的时候有什么别的方法，也发到评论区分享一下吧。

最后，我给你留下一个思考题。前面我们在构造第一个例子的过程中，通过**session A**的配合，让**session B**删除数据后又重新插入了一遍数据，然后就发现**explain**结果中，**rows**字段从10001变成37000多。

而如果没有**session A**的配合，只是单独执行**delete from t**、**call idata()**、**explain**这三句话，会看到**rows**字段其实还是10000左右。你可以自己验证一下这个结果。

这是什么原因呢？也请你分析一下吧。

你可以把你的分析结论写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章最后留给你的问题是，如果某次写入使用了**change buffer**机制，之后主机异常重启，是否会丢失**change buffer**和数据。

这个问题的答案是不会丢失，留言区的很多同学都回答对了。虽然是只更新内存，但是在事务提交的时候，我们把**change buffer**的操作也记录到**redo log**里了，所以崩溃恢复的时候，**change buffer**也能找回来。

在评论区有同学问到，**merge**的过程是否会把数据直接写回磁盘，这是个好问题。这里，我再为你分析一下。

merge的执行流程是这样的：

1. 从磁盘读入数据页到内存（老版本的数据页）；
2. 从**change buffer**里找出这个数据页的**change buffer**记录（可能有多个），依次应用，得到新版数据页；
3. 写**redo log**。这个**redo log**包含了数据的变更和**change buffer**的变更。

到这里**merge**过程就结束了。这时候，数据页和内存中**change buffer**对应的磁盘位置都还没有修改，属于脏页，之后各自刷回自己的物理数据，就是另外一个过程了。

评论区留言点赞板：

@某、人 把02篇的**redo log**更新细节和**change buffer**的更新串了起来；
@lvan 回复了其他同学的问题，并联系到**Checkpoint**机制；
@约书亚 问到了**merge**和**redolog**的关系。

The image is a promotional graphic for a MySQL course. It features a portrait of a man with glasses and a black shirt on the right side. On the left, there's text about the course and the instructor. The text includes the course title 'MySQL 实战 45 讲', the subtitle '从原理到实战，丁奇带你搞懂 MySQL', the instructor's name '林晓斌' (alias '丁奇'), and their title '前阿里资深技术专家'. The 'GeekTime' logo is also present.

极客时间

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌 网名丁奇 前阿里资深技术专家

11 | 怎么给字符串字段加索引？

2018-12-07 林晓斌



现在，几乎所有的系统都支持邮箱登录，如何在邮箱这样的字段上建立合理的索引，是我们今天要讨论的问题。

假设，你现在维护一个支持邮箱登录的系统，用户表是这么定义的：

```
mysql> create table SUser(
    ID bigint unsigned primary key,
    email varchar(64),
    ...
)engine=innodb;
```

由于要使用邮箱登录，所以业务代码中一定会出现类似于这样的语句：

```
mysql> select f1, f2 from SUser where email='xxx';
```

从第4和第5篇讲解索引的文章中，我们可以知道，如果email这个字段上没有索引，那么这个语句就只能做全表扫描。

同时，MySQL是支持前缀索引的，也就是说，你可以定义字符串的一部分作为索引。默认地，

如果你创建索引的语句不指定前缀长度，那么索引就会包含整个字符串。

比如，这两个在email字段上创建索引的语句：

```
mysql> alter table SUser add index index1(email);  
或  
mysql> alter table SUser add index index2(email(6));
```

第一个语句创建的**index1**索引里面，包含了每个记录的整个字符串；而第二个语句创建的**index2**索引里面，对于每个记录都是只取前6个字节。

那么，这两种不同的定义在数据结构和存储上有什么区别呢？如图2和3所示，就是这两个索引的示意图。

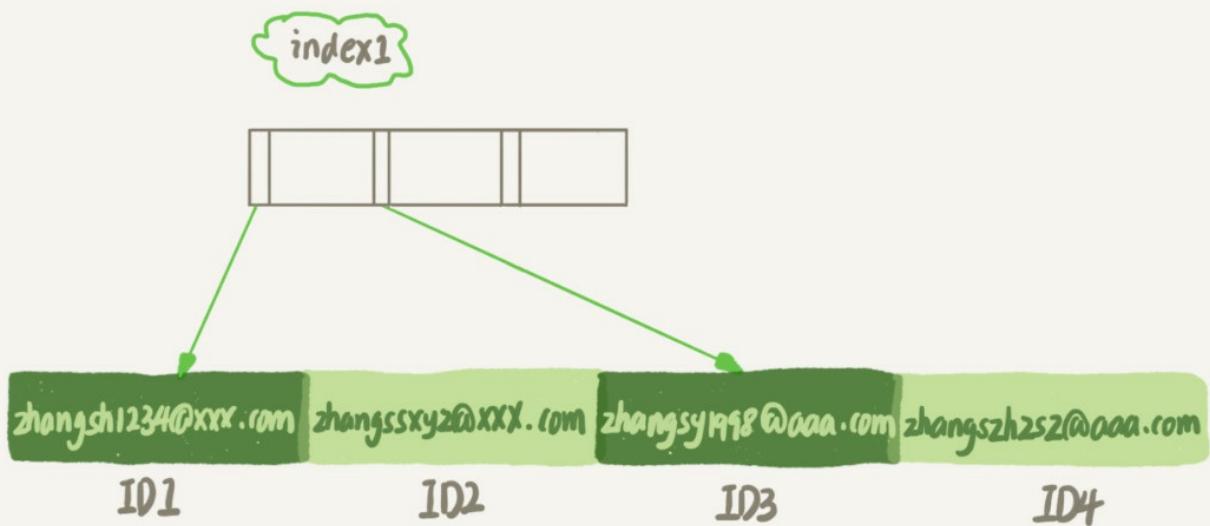


图1 email 索引结构

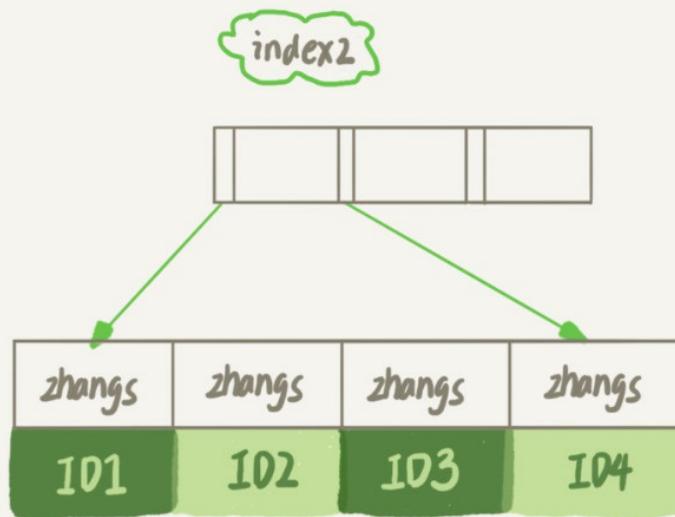


图2 email(6) 索引结构

从图中你可以看到，由于email(6)这个索引结构中每个邮箱字段都只取前6个字符（即：zhangs），所以占用的空间会更小，这就是使用前缀索引的优势。

但，这同时带来的损失是，可能会增加额外的记录扫描次数。

接下来，我们再看看下面这个语句，在这两个索引定义下分别是怎么执行的。

```
select id,name,email from SUser where email='zhangssxyz@xxx.com';
```

如果使用的是index1（即email整个字符串的索引结构），执行顺序是这样的：

1. 从index1索引树找到满足索引值是'zhangssxyz@xxx.com'的这条记录，取得ID2的值；
2. 到主键上查到主键值是ID2的行，判断email的值是正确的，将这行记录加入结果集；
3. 取index1索引树上刚刚查到的位置的下一条记录，发现已经不满足email='zhangssxyz@xxx.com'的条件了，循环结束。

这个过程中，只需要回主键索引取一次数据，所以系统认为只扫描了一行。

如果使用的是**index2**（即**email(6)**索引结构），执行顺序是这样的：

1. 从**index2**索引树找到满足索引值是'**zhangs**'的记录，找到的第一个是**ID1**；
2. 到主键上查到主键值是**ID1**的行，判断出**email**的值不是'**zhangssxyz@xxx.com**'，这行记录丢弃；
3. 取**index2**上刚刚查到的位置的下一条记录，发现仍然是'**zhangs**'，取出**ID2**，再到**ID**索引上取整行然后判断，这次值对了，将这行记录加入结果集；
4. 重复上一步，直到在**index2**上取到的值不是'**zhangs**'时，循环结束。

在这个过程中，要回主键索引取4次数据，也就是扫描了4行。

通过这个对比，你很容易就可以发现，使用前缀索引后，可能会导致查询语句读数据的次数变多。

但是，对于这个查询语句来说，如果你定义的**index2**不是**email(6)**而是**email(7)**，也就是说取**email**字段的前7个字节来构建索引的话，即满足前缀'**zhangss**'的记录只有一个，也能够直接查到**ID2**，只扫描一行就结束了。

也就是说使用前缀索引，定义好长度，就可以做到既节省空间，又不用额外增加太多的查询成本。

于是，你就有个问题：当要给字符串创建前缀索引时，有什么方法能够确定我应该使用多长的前缀呢？

实际上，我们在建立索引时关注的是区分度，区分度越高越好。因为区分度越高，意味着重复的键值越少。因此，我们可以通过统计索引上有多少个不同的值来判断要使用多长的前缀。

首先，你可以使用下面这个语句，算出这个列上有多少个不同的值：

```
mysql> select count(distinct email) as L from SUser;
```

然后，依次选取不同长度的前缀来看这个值，比如我们要看一下4~7个字节的前缀索引，可以用这个语句：

```
mysql> select
    count(distinct left(email,4)) as L4,
    count(distinct left(email,5)) as L5,
    count(distinct left(email,6)) as L6,
    count(distinct left(email,7)) as L7,
  from SUser;
```

当然，使用前缀索引很可能会损失区分度，所以你需要预先设定一个可以接受的损失比例，比如5%。然后，在返回的L4~L7中，找出不小于 $L * 95\%$ 的值，假设这里L6、L7都满足，你就可以选择前缀长度为6。

前缀索引对覆盖索引的影响

前面我们说了使用前缀索引可能会增加扫描行数，这会影响到性能。其实，前缀索引的影响不止如此，我们再看一下另外一个场景。

你先来看看这个SQL语句：

```
select id,email from SUser where email='zhangssxyz@xxx.com';
```

与前面例子中的SQL语句

```
select id,name,email from SUser where email='zhangssxyz@xxx.com';
```

相比，这个语句只要求返回id和email字段。

所以，如果使用**index1**（即email整个字符串的索引结构）的话，可以利用覆盖索引，从**index1**查到结果后直接就返回了，不需要回到ID索引再去查一次。而如果使用**index2**（即email(6)索引结构）的话，就不得不回到ID索引再去判断email字段的值。

即使你将**index2**的定义修改为email(18)的前缀索引，这时候虽然**index2**已经包含了所有的信息，但InnoDB还是要回到id索引再查一下，因为系统并不确定前缀索引的定义是否截断了完整信息。

也就是说，使用前缀索引就用不上覆盖索引对查询性能的优化了，这也是你在选择是否使用前缀索引时需要考虑的一个因素。

其他方式

对于类似于邮箱这样的字段来说，使用前缀索引的效果可能还不错。但是，遇到前缀的区分度不

够好的情况时，我们要怎么办呢？

比如，我们国家的身份证号，一共**18位**，其中前**6位**是地址码，所以同一个县的人的身份证号前**6位**一般会是相同的。

假设你维护的数据库是一个市的公民信息系统，这时候如果对身份证号做长度为**6**的前缀索引的话，这个索引的区分度就非常低了。

按照我们前面说的方法，可能你需要创建长度为**12**以上的前缀索引，才能够满足区分度要求。

但是，索引选取的越长，占用的磁盘空间就越大，相同的数据页能放下的索引值就越少，搜索的效率也就会越低。

那么，如果我们能够确定业务需求里面只有按照身份证进行等值查询的需求，还有没有别的处理方法呢？这种方法，既可以占用更小的空间，也能达到相同的查询效率。

答案是，有的。

第一种方式是使用倒序存储。如果你存储身份证号的时候把它倒过来存，每次查询的时候，你可以这么写：

```
mysql> select field_list from t where id_card = reverse('input_id_card_string');
```

由于身份证号的最后**6位**没有地址码这样的重复逻辑，所以最后这**6位**很可能就提供了足够的区分度。当然了，实践中你不要忘记使用**count(distinct)**方法去做个验证。

第二种方式是使用hash字段。你可以在表上再创建一个整数字段，来保存身份证的校验码，同时在这个字段上创建索引。

```
mysql> alter table t add id_card_crc int unsigned, add index(id_card_crc);
```

然后每次插入新记录的时候，都同时用**crc32()**这个函数得到校验码填到这个新字段。由于校验码可能存在冲突，也就是说两个不同的身份证号通过**crc32()**函数得到的结果可能是相同的，所以你的查询语句**where**部分要判断**id_card**的值是否精确相同。

```
mysql> select field_list from t where id_card_crc=crc32('input_id_card_string') and id_card='i
```

这样，索引的长度变成了**4**个字节，比原来小了很多。

接下来，我们再一起看看使用倒序存储和使用**hash**字段这两种方法的异同点。

首先，它们的相同点是，都不支持范围查询。倒序存储的字段上创建的索引是按照倒序字符串的方式排序的，已经没有办法利用索引方式查出身份证号码在[ID_X, ID_Y]的所有市民了。同样地，**hash**字段的方式也只能支持等值查询。

它们的区别，主要体现在以下三个方面：

1. 从占用的额外空间来看，倒序存储方式在主键索引上，不会消耗额外的存储空间，而**hash**字段方法需要增加一个字段。当然，倒序存储方式使用4个字节的前缀长度应该是不够的，如果再长一点，这个消耗跟额外这个**hash**字段也差不多抵消了。
2. 在CPU消耗方面，倒序方式每次写和读的时候，都需要额外调用一次**reverse**函数，而**hash**字段的方式需要额外调用一次**crc32()**函数。如果只从这两个函数的计算复杂度来看的话，**reverse**函数额外消耗的CPU资源会更小些。
3. 从查询效率上看，使用**hash**字段方式的查询性能相对更稳定一些。因为**crc32**算出来的值虽然有冲突的概率，但是概率非常小，可以认为每次查询的平均扫描行数接近1。而倒序存储方式毕竟还是用的前缀索引的方式，也就是说还是会增加扫描行数。

小结

在今天这篇文章中，我跟你聊了聊字符串字段创建索引的场景。我们来回顾一下，你可以使用的方式有：

1. 直接创建完整索引，这样可能比较占用空间；
2. 创建前缀索引，节省空间，但会增加查询扫描次数，并且不能使用覆盖索引；
3. 倒序存储，再创建前缀索引，用于绕过字符串本身前缀的区分度不够的问题；
4. 创建**hash**字段索引，查询性能稳定，有额外的存储和计算消耗，跟第三种方式一样，都不支持范围扫描。

在实际应用中，你要根据业务字段的特点选择使用哪种方式。

好了，又到了最后的问题时间。

如果你在维护一个学校的学生信息数据库，学生登录名的统一格式是"学号@gmail.com"，而学号的规则是：十五位的数字，其中前三位是所在城市编号、第四到第六位是学校编号、第七位到第十位是入学年份、最后五位是顺序编号。

系统登录的时候都需要学生输入登录名和密码，验证正确后才能继续使用系统。就只考虑登录验证这个行为的话，你会怎么设计这个登录名的索引呢？

你可以把你的分析思路和设计结果写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上篇文章中的第一个例子，评论区有几位同学说没有复现，大家要检查一下隔离级别是不是**RR**（**Repeatable Read**, 可重复读），创建的表**t**是不是**InnoDB**引擎。我把复现过程做成了一个视频，供你参考。

在上一篇文章最后，我给你留的问题是，为什么经过这个操作序列，**explain**的结果就不对了？这里，我来为你分析一下原因。

delete语句删掉了所有的数据，然后再通过**call idata()**插入了**10万行**数据，看上去是覆盖了原来的**10万行**。

但是，**session A**开启了事务并没有提交，所以之前插入的**10万行**数据是不能删除的。这样，之前的数据每一行数据都有两个版本，旧版本是**delete**之前的数据，新版本是标记为**deleted**的数据。

这样，索引**a**上的数据其实就有两份。

然后你会说，不对啊，主键上的数据也不能删，那没有使用**force index**的语句，使用**explain**命令看到的扫描行数为什么还是**100000**左右？（潜台词，如果这个也翻倍，也许优化器还会认为选字段**a**作为索引更合适）

是的，不过这个是主键，主键是直接按照表的行数来估计的。而表的行数，优化器直接用的是**show table status**的值。

这个值的计算方法，我会在后面有文章为你详细讲解。

```

mysql> explain select * from t where a between 10000 and 20000;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | t      | NULL       | ALL  | a             | NULL | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+
rows   filtered | Extra
104128    50.00 | Using where
1 row in set, 1 warning (0.01 sec)

mysql> show table status like 't'\G
***** 1. row *****
      Name: t
      Engine: InnoDB
      Version: 10
      Row_format: Dynamic
      Rows: 104128
      Avg_row_length: 88
      Data_length: 8929280
      Max_data_length: 0
      Index_length: 13664256
      Data_free: 4194384
      Auto_increment: 300001
      Create_time: 2018-12-06 17:45:49
      Update_time: 2018-12-06 17:48:55
      Check_time: NULL
      Collation: latin1_swedish_ci
      Checksum: NULL
      Create_options:
      Comment:
1 row in set (0.00 sec)

```

评论区留言点赞板：

@斜面镜子 Bill 的评论最接近答案；

@某、人 做了两个很不错的对照试验；

@ye7zi 等几位同学很认真的验证，赞态度。大家的机器如果IO能力比较差的话，做这个验证的时候，可以把`innodb_flush_log_at_trx_commit` 和 `sync_binlog` 都设置成0。

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家

12 | 为什么我的MySQL会“抖”一下？

2018-12-10 林晓斌



平时的工作中，不知道你有没有遇到过这样的场景，一条SQL语句，正常执行的时候特别快，但是有时也不知道怎么回事，它就会变得特别慢，并且这样的场景很难复现，它不只随机，而且持续时间还很短。

看上去，这就像是数据库“抖”了一下。今天，我们就一起来看一看这是什么原因。

你的SQL语句为什么变“慢”了

在前面第2篇文章 [《日志系统：一条SQL更新语句是如何执行的？》](#) 中，我为你介绍了WAL机制。现在你知道了，InnoDB在处理更新语句的时候，只做了写日志这一个磁盘操作。这个日志叫作redo log（重做日志），也就是《孔乙己》里咸亨酒店掌柜用来记账的粉板，在更新内存写完redo log后，就返回给客户端，本次更新成功。

做下类比的话，掌柜记账的账本是数据文件，记账用的粉板是日志文件（redo log），掌柜的记忆就是内存。

掌柜总要找时间把账本更新一下，这对应的就是把内存里的数据写入磁盘的过程，术语就是flush。在这个flush操作执行之前，孔乙己的赊账总额，其实跟掌柜手中账本里面的记录是不一致的。因为孔乙己今天的赊账金额还只在粉板上，而账本里的记录是老的，还没把今天的赊账算进去。

当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。内存数据写

入到磁盘后，内存和磁盘上的数据页的内容就一致了，称为“干净页”。

不论是脏页还是干净页，都在内存中。在这个例子里，内存对应的就是掌柜的记忆。

接下来，我们用一个示意图来展示一下“孔乙己赊账”的整个操作过程。假设原来孔乙己欠账**10**文，这次又要赊**9**文。

更新/账本过程

日志/粉板

10改成19

内存/掌柜记忆



磁盘/账本



flush/改账本

日志/粉板

~~10改成19~~

内存/掌柜记忆



磁盘/账本



图1 “孔乙己账本”更新和flush过程

回到文章开头的问题，你不难想象，平时执行很快的更新操作，其实就是在写内存和日志，而MySQL偶尔“抖”一下的那个瞬间，可能就是在刷脏页（flush）。

那么，什么情况会引发数据库的**flush**过程呢？

我们还是继续用咸亨酒店掌柜的例子，想一想：掌柜在什么情况下会把粉板上的赊账记录改到账本上？

- 第一种场景是，粉板满了，记不下了。这时候如果再有人来赊账，掌柜就只得放下手里的活儿，将粉板上的记录擦掉一些，留出空位以便继续记账。当然在擦掉之前，他必须先将正确的账目记录到账本中才行。

这个场景，对应的就是InnoDB的**redo log**写满了。这时候系统会停止所有更新操作，把**checkpoint**往前推进，**redo log**留出空间可以继续写。我在第二讲画了一个**redo log**的示意图，这里我改成环形，便于大家理解。

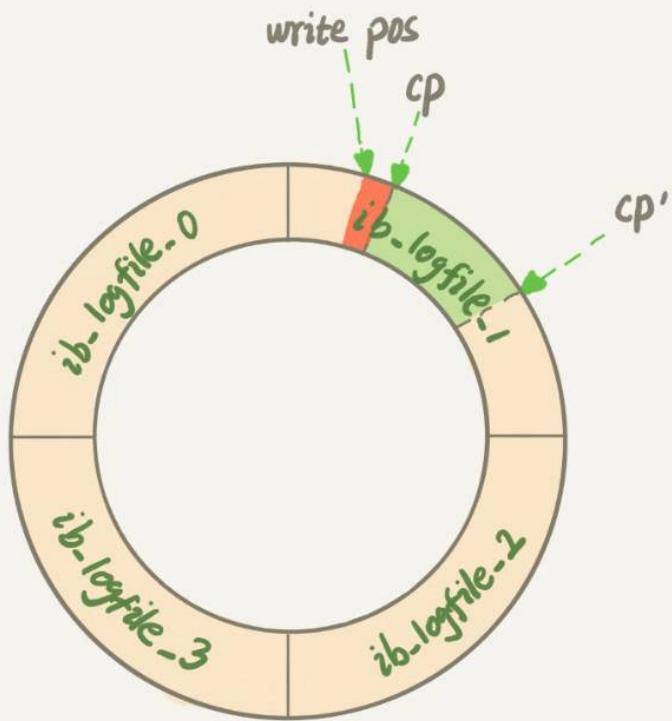


图2 redo log状态图

checkpoint可不是随便往前修改一下位置就可以的。比如图2中，把**checkpoint**位置从**CP**推进到**CP'**，就需要将两个点之间的日志（浅绿色部分），对应的所有脏页都**flush**到磁盘上。之后，图中从**write pos**到**CP'**之间就是可以再写入的**redo log**的区域。

- 第二种场景是，这一天生意太好，要记住的事情太多，掌柜发现自己快记不住了，赶紧找出账本把孔乙己这笔账先加进去。

这种场景，对应的就是系统内存不足。当需要新的内存页，而内存不够用的时候，就要淘汰

一些数据页，空出内存给别的数据页使用。如果淘汰的是“脏页”，就要先将脏页写到磁盘。你一定会说，这时候难道不能直接把内存淘汰掉，下次需要请求的时候，从磁盘读入数据页，然后拿**redo log**出来应用不就行了？这里其实是从性能考虑的。如果刷脏页一定会写盘，就保证了每个数据页有两种状态：

- 一种是内存里存在，内存里就肯定是正确的结果，直接返回；
 - 另一种是内存里没有数据，就可以肯定数据文件上是正确的结果，读入内存后返回。
- 这样的效率最高。
- 第三种场景是，生意不忙的时候，或者打烊之后。这时候柜台没事，掌柜闲着也是闲着，不如更新账本。
- 这种场景，对应的就是**MySQL**认为系统“空闲”的时候。当然，**MySQL**“这家酒店”的生意好起来可是会很快就能把粉板记满的，所以“掌柜”要合理地安排时间，即使是“生意好”的时候，也要见缝插针地找时间，只要有机会就刷一点“脏页”。
- 第四种场景是，年底了咸亨酒店要关门几天，需要把账结清一下。这时候掌柜要把所有账都记到账本上，这样过完年重新开张的时候，就能就着账本明确账目情况了。
- 这种场景，对应的就是**MySQL**正常关闭的情况。这时候，**MySQL**会把内存的脏页都**flush**到磁盘上，这样下次**MySQL**启动的时候，就可以直接从磁盘上读数据，启动速度会很快。

接下来，你可以分析一下上面四种场景对性能的影响。

其中，第三种情况是属于**MySQL**空闲时的操作，这时系统没什么压力，而第四种场景是数据库本来就要关闭了。这两种情况下，你不会太关注“性能”问题。所以这里，我们主要来分析一下前两种场景下的性能问题。

第一种是“**redo log**写满了，要**flush**脏页”，这种情况是**InnoDB**要尽量避免的。因为出现这种情况的时候，整个系统就不能再接受更新了，所有的更新都必须堵住。如果你从监控上看，这时候更新数会跌为0。

第二种是“内存不够用了，要先将脏页写到磁盘”，这种情况其实是常态。**InnoDB用缓冲池**（**buffer pool**）管理内存，缓冲池中的内存页有三种状态：

- 第一种是，还没有使用的；
- 第二种是，使用了并且是干净页；
- 第三种是，使用了并且是脏页。

InnoDB的策略是尽量使用内存，因此对于一个长时间运行的库来说，未被使用的页面很少。

而当要读入的数据页没有在内存的时候，就必须到缓冲池中申请一个数据页。这时候只能把最久不使用的数据页从内存中淘汰掉：如果要淘汰的是一个干净页，就直接释放出来复用；但如果是

脏页呢，就必须将脏页先刷到磁盘，变成干净页后才能复用。

所以，刷脏页虽然是常态，但是出现以下这两种情况，都是会明显影响性能的：

1. 一个查询要淘汰的脏页个数太多，会导致查询的响应时间明显变长；
2. 日志写满，更新全部堵住，写性能跌为0，这种情况对敏感业务来说，是不能接受的。

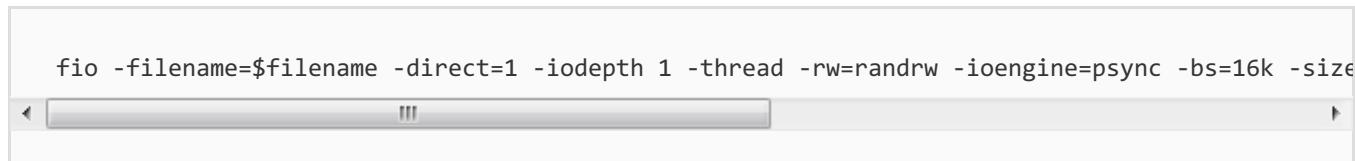
所以，InnoDB需要有控制脏页比例的机制，来尽量避免上面的这两种情况。

InnoDB刷脏页的控制策略

接下来，我就来和你说说InnoDB脏页的控制策略，以及和这些策略相关的参数。

首先，你要正确地告诉InnoDB所在主机的IO能力，这样InnoDB才能知道需要全力刷脏页的时候，可以刷多快。

这就要用到`innodb_io_capacity`这个参数了，它会告诉InnoDB你的磁盘能力。这个值我建议你设置成磁盘的IOPS。磁盘的IOPS可以通过fio这个工具来测试，下面的语句是我用来测试磁盘随机读写的命令：



```
fio -filename=$filename -direct=1 -iodepth 1 -thread -rw=randrw -ioengine=psync -bs=16k -size
```

其实，因为没能正确地设置`innodb_io_capacity`参数，而导致的性能问题也比比皆是。之前，就曾有其他公司的开发负责人找我看一个库的性能问题，说MySQL的写入速度很慢，TPS很低，但是数据库主机的IO压力并不大。经过一番排查，发现罪魁祸首就是这个参数的设置出了问题。

他的主机磁盘用的是SSD，但是`innodb_io_capacity`的值设置的是300。于是，InnoDB认为这个系统的能力就这么差，所以刷脏页刷得特别慢，甚至比脏页生成的速度还慢，这样就造成了脏页累积，影响了查询和更新性能。

虽然我们现在已经定义了“全力刷脏页”的行为，但平时总不能一直是全力刷吧？毕竟磁盘能力不能只用来刷脏页，还需要服务用户请求。所以接下来，我们就一起看看InnoDB怎么控制引擎按照“全力”的百分比来刷脏页。

根据我前面提到的知识点，试想一下，如果你来设计策略控制刷脏页的速度，会参考哪些因素呢？

这个问题可以这么想，如果刷太慢，会出现什么情况？首先是内存脏页太多，其次是redo log写满。

所以，InnoDB的刷盘速度就是要参考这两个因素：一个是脏页比例，一个是redo log写盘速度。

InnoDB会根据这两个因素先单独算出两个数字。

参数`innodb_max_dirty_pages_pct`是脏页比例上限，默认值是75%。InnoDB会根据当前的脏页比例（假设为M），算出一个范围在0到100之间的数字，计算这个数字的伪代码类似这样：

```
F1(M)
{
    if M>=innodb_max_dirty_pages_pct then
        return 100;
    return 100*M/innodb_max_dirty_pages_pct;
}
```

InnoDB每次写入的日志都有一个序号，当前写入的序号跟checkpoint对应的序号之间的差值，我们假设为N。InnoDB会根据这个N算出一个范围在0到100之间的数字，这个计算公式可以记为F2(N)。F2(N)算法比较复杂，你只要知道N越大，算出来的值越大就好了。

然后，根据上述算得的F1(M)和F2(N)两个值，取其中较大的值记为R，之后引擎就可以按照`innodb_io_capacity`定义的能力乘以R%来控制刷脏页的速度。

上述的计算流程比较抽象，不容易理解，所以我画了一个简单的流程图。图中的F1、F2就是上面我们通过脏页比例和redo log写入速度算出来的两个值。

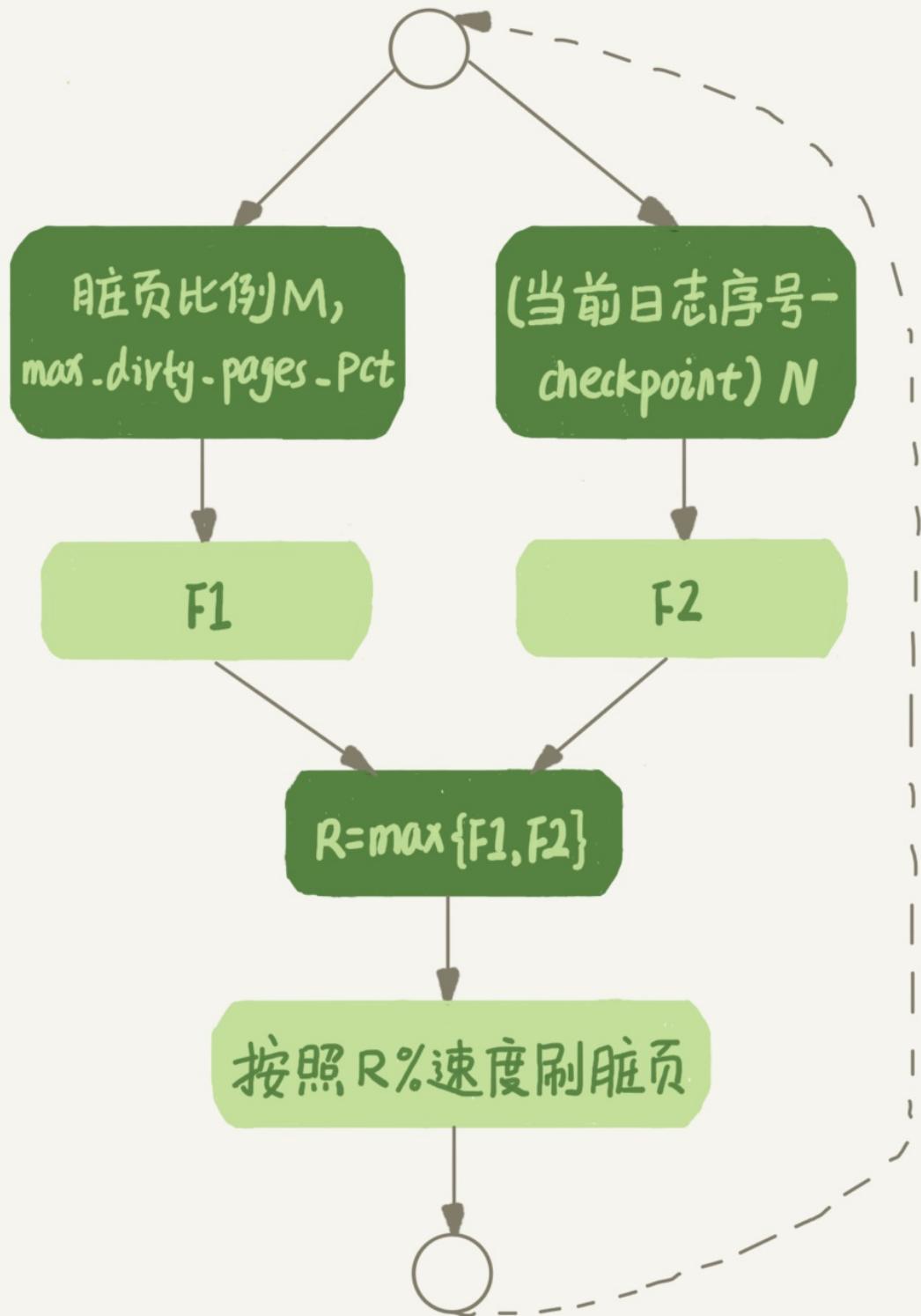


图3 InnoDB刷脏页速度策略

现在你知道了，InnoDB会在后台刷脏页，而刷脏页的过程是要将内存页写入磁盘。所以，无论是你的查询语句在需要内存的时候可能要求淘汰一个脏页，还是由于刷脏页的逻辑会占用IO资源并可能影响到了你的更新语句，都可能是造成你从业务端感知到MySQL“抖”了一下的原因。

要尽量避免这种情况，你就要合理地设置`innodb_io_capacity`的值，并且平时要多关注脏页比例，不要让它经常接近75%。

其中，脏页比例是通过`Innodb_buffer_pool_pages_dirty/Innodb_buffer_pool_pages_total`得到的，具体的命令参考下面的代码：

```
mysql> select VARIABLE_VALUE into @a from global_status where VARIABLE_NAME = 'Innodb_buffer_pool_pages_dirty';
mysql> select VARIABLE_VALUE into @b from global_status where VARIABLE_NAME = 'Innodb_buffer_pool_pages_total';
mysql> select @a/@b;
```

接下来，我们再看一个有趣的策略。

一旦一个查询请求需要在执行过程中先`flush`掉一个脏页时，这个查询就可能要比平时慢了。而MySQL中的一个机制，可能让你的查询会更慢：在准备刷一个脏页的时候，如果这个数据页旁边的数据页刚好是脏页，就会把这个“邻居”也带着一起刷掉；而且这个把“邻居”拖下水的逻辑还可以继续蔓延，也就是对于每个邻居数据页，如果跟它相邻的数据页也还是脏页的话，也会被放到一起刷。

在InnoDB中，`innodb_flush_neighbors`参数就是用来控制这个行为的，值为1的时候会有上述的“连坐”机制，值为0时表示不找邻居，自己刷自己的。

找“邻居”这个优化在机械硬盘时代是很有意义的，可以减少很多随机IO。机械硬盘的随机IOPS一般只有几百，相同的逻辑操作减少随机IO就意味着系统性能的大幅度提升。

而如果使用的是SSD这类IOPS比较高的设备的话，我就建议你把`innodb_flush_neighbors`的值设置成0。因为这时候IOPS往往不是瓶颈，而“只刷自己”，就能更快地执行完必要的刷脏页操作，减少SQL语句响应时间。

在MySQL 8.0中，`innodb_flush_neighbors`参数的默认值已经是0了。

小结

今天这篇文章，我延续第2篇中介绍的WAL的概念，和你解释了这个机制后续需要的刷脏页操作和执行时机。利用WAL技术，数据库将随机写转换成了顺序写，大大提升了数据库的性能。

但是，由此也带来了内存脏页的问题。脏页会被后台线程自动`flush`，也会由于数据页淘汰而触发`flush`，而刷脏页的过程由于会占用资源，可能会让你的更新和查询语句的响应时间长一些。在文章里，我也给你介绍了控制刷脏页的方法和对应的监控方式。

文章最后，我给你留下一个思考题吧。

一个内存配置为128GB、`innodb_io_capacity`设置为20000的大规格实例，正常会建议你将`redo log`设置成4个1GB的文件。

但如果你在配置的时候不慎将`redo log`设置成了1个100M的文件，会发生什么情况呢？又为什么会出现这样的情况呢？

你可以把你的分析结论写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期我留给你的问题是，给一个学号字段创建索引，有哪些方法。

由于这个学号的规则，无论是正向还是反向的前缀索引，重复度都比较高。因为维护的只是一个学校的，因此前面6位（其中，前三位是所在城市编号、第四到第六位是学校编号）其实是固定的，邮箱后缀都是@gamil.com，因此可以只存入学年份加顺序编号，它们的长度是9位。

而其实在此基础上，可以用数字类型来存这9位数字。比如201100001，这样只需要占4个字节。其实这个就是一种hash，只是它用了最简单的转换规则：字符串转数字的规则，而刚好我们设定的这个背景，可以保证这个转换后结果的唯一性。

评论区中，也有其他一些很不错的见解。

评论用户@封建的风说，一个学校的总人数这种数据量，50年才100万学生，这个表肯定是小表。为了业务简单，直接存原来的字符串。这个答复里面包含了“优化成本和收益”的思想，我觉得值得at出来。

@小潘同学提了另外一个极致的方向。如果碰到表数据量特别大的场景，通过这种方式的收益是很不错的。

评论区留言点赞板：

@ltzzll，提到了用整型存“四位年份+五位编号”的方法；

由于整个学号的值超过了int上限，@老杨同志也提到了用8个字节的bigint来存的方法。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



13 | 为什么表数据删掉一半，表文件大小不变？

2018-12-12 林晓斌



经常会有同学来问我，我的数据库占用空间太大，我把一个最大的表删掉了一半的数据，怎么表文件的大小还是没变？

那么今天，我就和你聊聊数据库表的空间回收，看看如何解决这个问题。

这里，我们还是针对MySQL中应用最广泛的InnoDB引擎展开讨论。一个InnoDB表包含两部分，即：表结构定义和数据。在MySQL 8.0版本以前，表结构是存在以.frm为后缀的文件里。而MySQL 8.0版本，则已经允许把表结构定义放在系统数据表中了。因为表结构定义占用的空间很小，所以我们今天主要讨论的是表数据。

接下来，我会先和你说明为什么简单地删除表数据达不到表空间回收的效果，然后再和你介绍正确回收空间的方法。

参数innodb_file_per_table

表数据既可以存在共享表空间里，也可以是单独的文件。这个行为是由参数innodb_file_per_table控制的：

1. 这个参数设置为OFF表示的是，表的数据放在系统共享表空间，也就是跟数据字典放在一起；
2. 这个参数设置为ON表示的是，每个InnoDB表数据存储在一个以.ibd为后缀的文件中。

从MySQL 5.6.6版本开始，它的默认值就是ON了。

我建议你不论使用MySQL的哪个版本，都将这个值设置为ON。因为，一个表单独存储为一个文件更容易管理，而且在你不需要这个表的时候，通过`drop table`命令，系统就会直接删除这个文件。而如果是放在共享表空间中，即使表删掉了，空间也是不会回收的。

所以，将`innodb_file_per_table`设置为ON，是推荐做法，我们接下来的讨论都是基于这个设置展开的。

我们在删除整个表的时候，可以使用`drop table`命令回收表空间。但是，我们遇到的更多的删除数据的场景是删除某些行，这时就遇到了我们文章开头的问题：表中的数据被删除了，但是表空间却没有被回收。

我们要彻底搞明白这个问题的话，就要从数据删除流程说起了。

数据删除流程

我们先再来看一下InnoDB中一个索引的示意图。在前面[第4](#)和[第5](#)篇文章中，我和你介绍索引时曾经提到过，InnoDB里的数据都是用B+树的结构组织的。

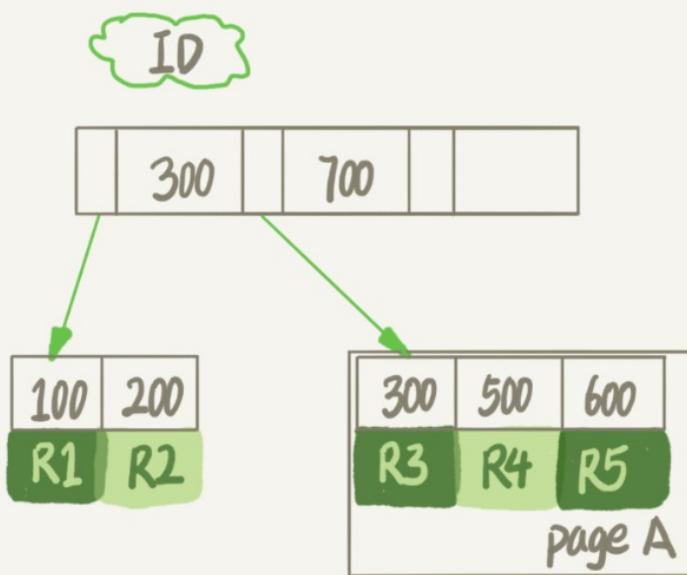


图1 B+树索引示意图

假设，我们要删掉R4这个记录，InnoDB引擎只会把R4这个记录标记为删除。如果之后要再插入一个ID在300和600之间的记录时，可能会复用这个位置。但是，磁盘文件的大小并不会缩小。

现在，你已经知道了InnoDB的数据是按页存储的，那么如果我们删掉了一个数据页上的所有记录，会怎么样？

答案是，整个数据页就可以被复用了。

但是，数据页的复用跟记录的复用是不同的。

记录的复用，只限于符合范围条件的数据。比如上面的这个例子，R4这条记录被删除后，如果插入一个ID是400的行，可以直接复用这个空间。但如果插入的是一个ID是800的行，就不能复用这个位置了。

而当整个页从B+树里面摘掉以后，可以复用到任何位置。以图1为例，如果将数据页page A上的所有记录删除以后，page A会被标记为可复用。这时候如果要插入一条ID=50的记录需要使用新页的时候，page A是可以被复用的。

如果相邻的两个数据页利用率都很小，系统就会把这个两个页上的数据合到其中一个页上，另外一个数据页就被标记为可复用。

进一步地，如果我们用delete命令把整个表的数据删除呢？结果就是，所有的数据页都会被标记为可复用。但是磁盘上，文件不会变小。

你现在知道了，delete命令其实只是把记录的位置，或者数据页标记为了“可复用”，但磁盘文件的大小是不会变的。也就是说，通过delete命令是不能回收表空间的。这些可以复用，而没有被使用的空间，看起来就像是“空洞”。

实际上，不止是删除数据会造成空洞，插入数据也会。

如果数据是按照索引递增顺序插入的，那么索引是紧凑的。但如果数据是随机插入的，就可能造成索引的数据页分裂。

假设图1中page A已经满了，这时我要再插入一行数据，会怎样呢？

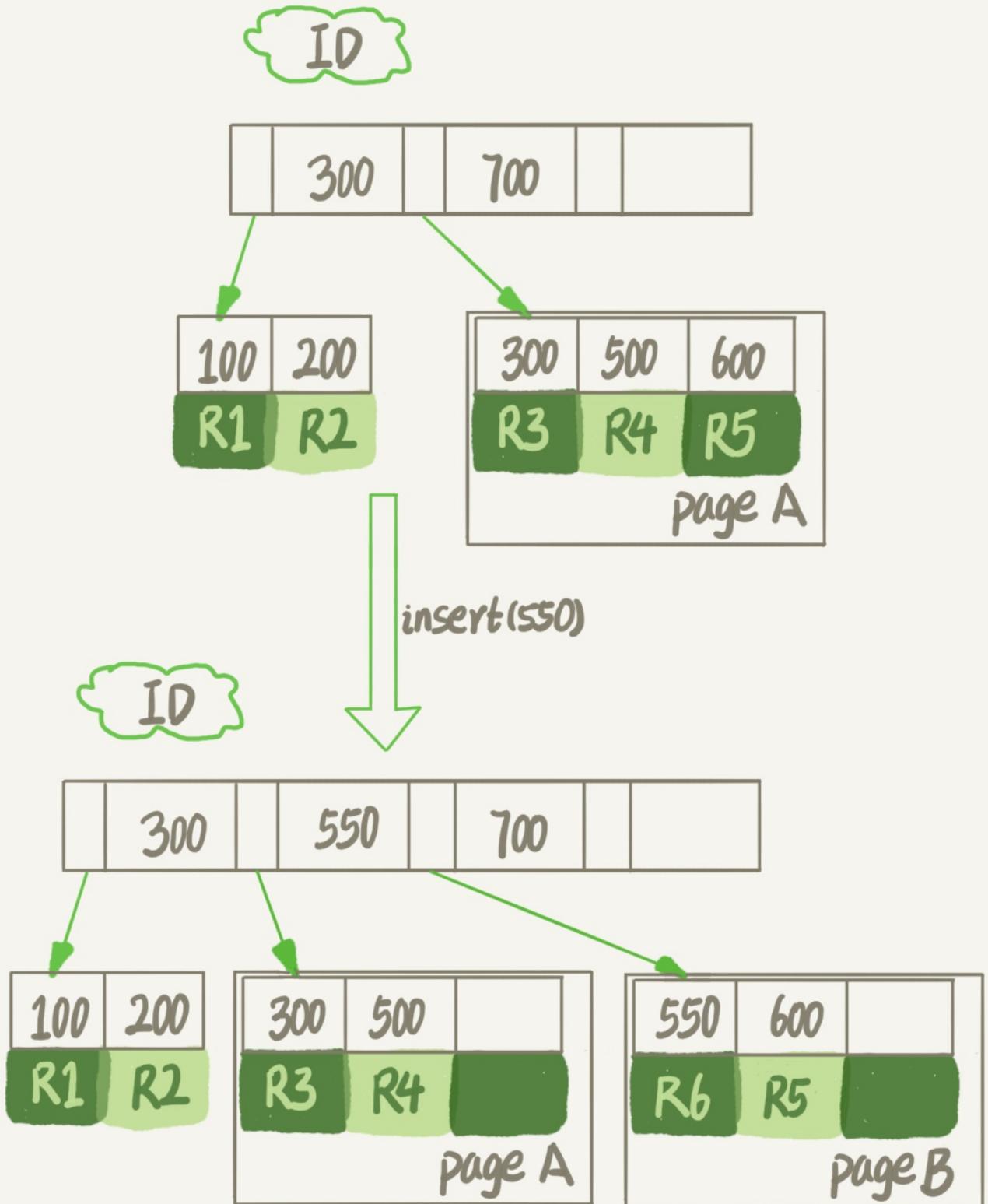


图2 插入数据导致页分裂

可以看到，由于**page A**满了，再插入一个ID是550的数据时，就不得不申请一个新的页面**page B**来保存数据了。页分裂完成后，**page A**的末尾就留下了空洞（注意：实际上，可能不止1个记录的位置是空洞）。

另外，更新索引上的值，可以理解为删除一个旧的值，再插入一个新值。不难理解，这也是会造成空洞的。

也就是说，经过大量增删改的表，都是可能是存在空洞的。所以，如果能够把这些空洞去掉，就能达到收缩表空间的目的。

而重建表，就可以达到这样的目的。

重建表

试想一下，如果你现在有一个表A，需要做空间收缩，为了把表中存在的空洞去掉，你可以怎么做呢？

你可以新建一个与表A结构相同的表B，然后按照主键ID递增的顺序，把数据一行一行地从表A里读出来再插入到表B中。

由于表B是新建的表，所以表A主键索引上的空洞，在表B中就都不存在了。显然地，表B的主键索引更紧凑，数据页的利用率也更高。如果我们把表B作为临时表，数据从表A导入表B的操作完成后，用表B替换A，从效果上看，就起到了收缩表A空间的作用。

这里，你可以使用`alter table A engine=InnoDB`命令来重建表。在MySQL 5.5版本之前，这个命令的执行流程跟我们前面描述的差不多，区别只是这个临时表B不需要你自己创建，MySQL会自动完成转存数据、交换表名、删除旧表的操作。

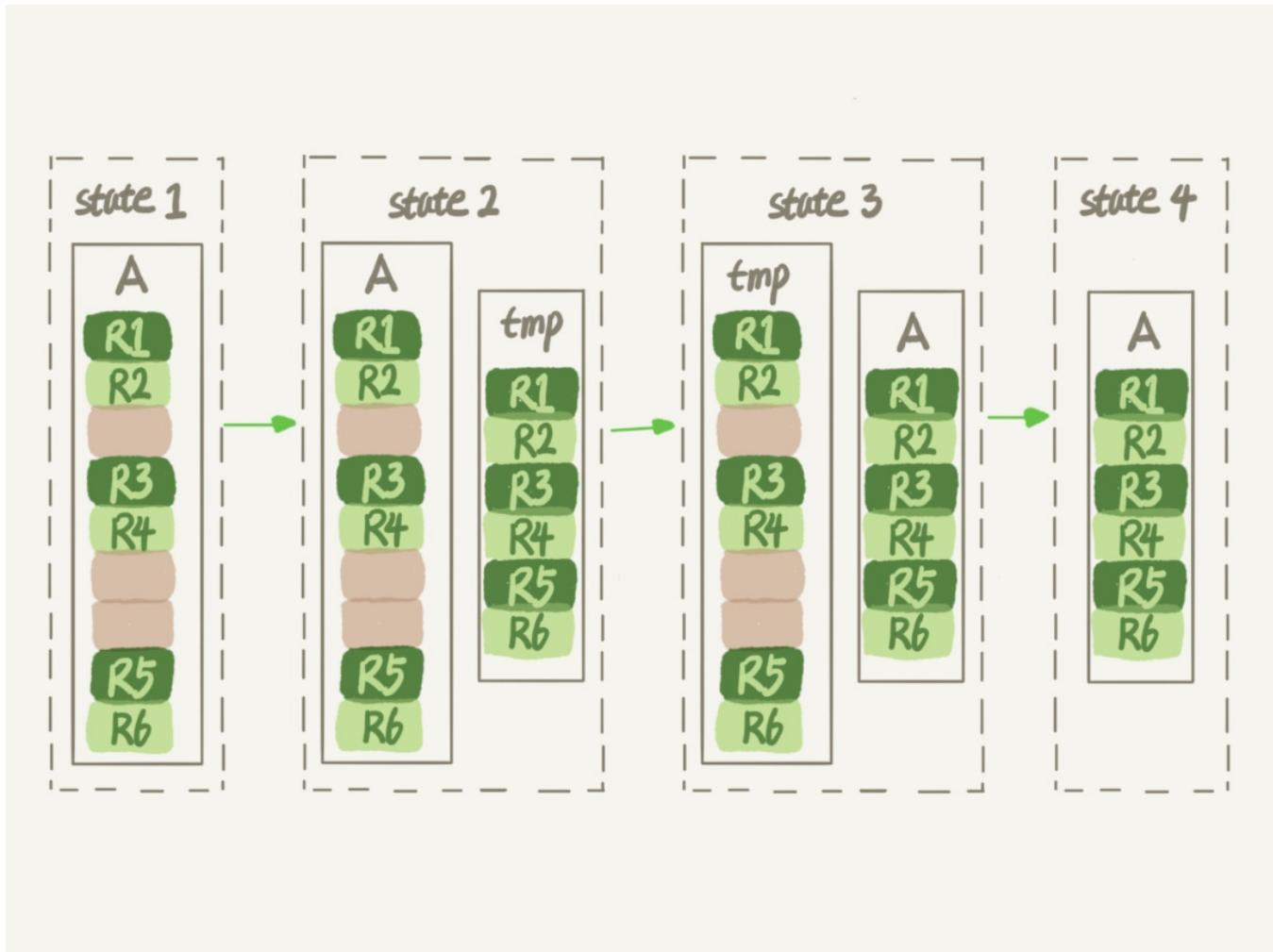


图3 改锁表DDL

显然，花时间最多的步骤是往临时表插入数据的过程，如果在这个过程中，有新的数据要写入到表A的话，就会造成数据丢失。因此，在整个**DDL**过程中，表A中不能有更新。也就是说，这个**DDL**不是**Online**的。

而在**MySQL 5.6**版本开始引入的**Online DDL**，对这个操作流程做了优化。

我给你简单描述一下引入了**Online DDL**之后，重建表的流程：

1. 建立一个临时文件，扫描表A主键的所有数据页；
2. 用数据页中表A的记录生成B+树，存储到临时文件中；
3. 生成临时文件的过程中，将所有对A的操作记录在一个日志文件（**row log**）中，对应的是图中**state2**的状态；
4. 临时文件生成后，将日志文件中的操作应用到临时文件，得到一个逻辑数据上与表A相同的数据文件，对应的就是图中**state3**的状态；
5. 用临时文件替换表A的数据文件。

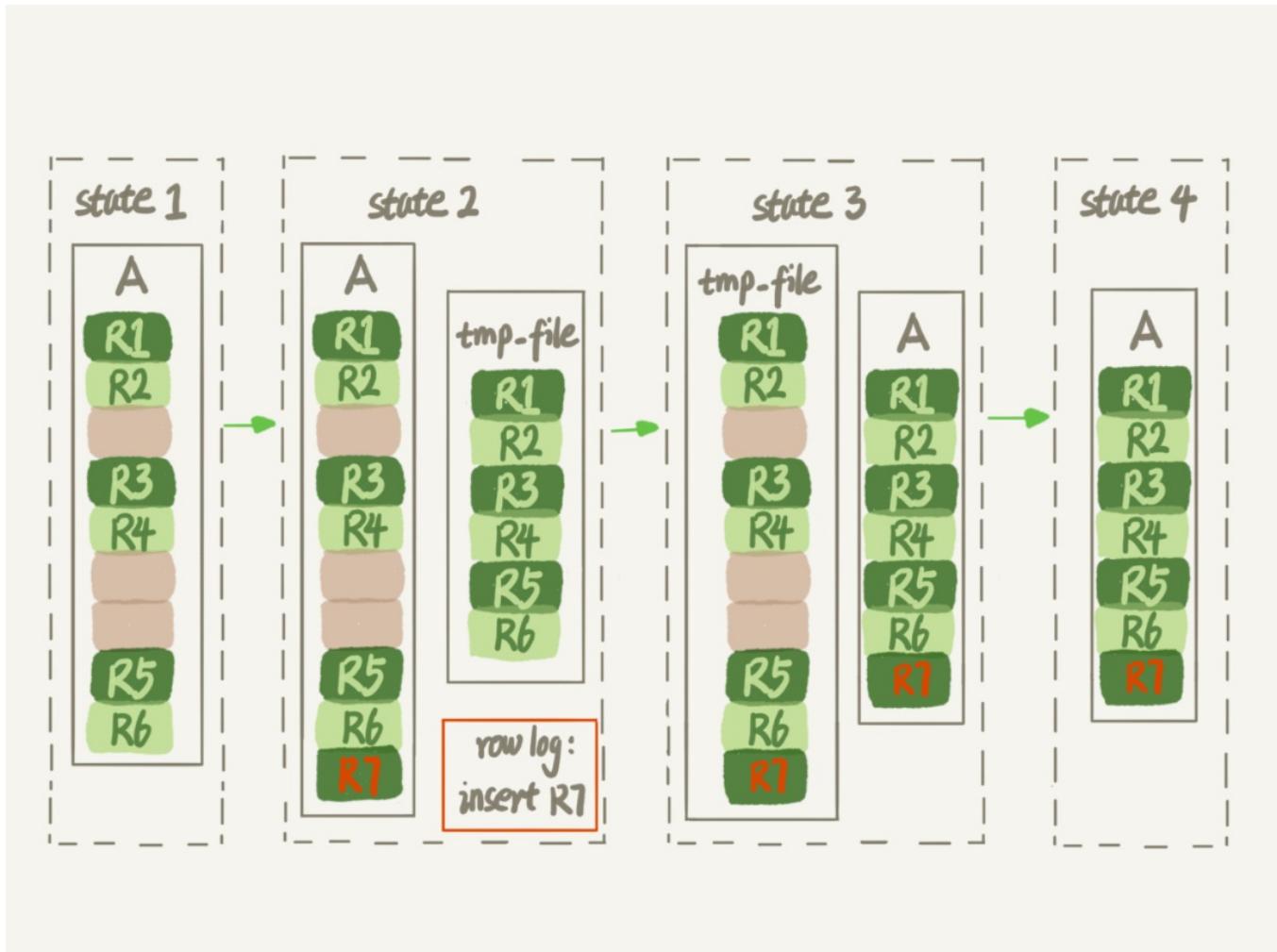


图4 Online DDL

可以看到，与图3过程的不同之处在于，由于日志文件记录和重放操作这个功能的存在，这个方案在重建表的过程中，允许对表A做增删改操作。这也就是**Online DDL**名字的来源。

我记得有同学在第6篇讲表锁的文章[《全局锁和表锁：给表加个字段怎么索这么多阻碍？》](#)的评论区留言说，**DDL**之前是要拿**MDL**写锁的，这样还能叫**Online DDL**吗？

确实，图4的流程中，**alter**语句在启动的时候需要获取**MDL**写锁，但是这个写锁在真正拷贝数据之前就退化成读锁了。

为什么要退化呢？为了实现**Online**，**MDL**读锁不会阻塞增删改操作。

那为什么不干脆直接解锁呢？为了保护自己，禁止其他线程对这个表同时做**DDL**。

而对于一个大表来说，**Online DDL**最耗时的过程就是拷贝数据到临时表的过程，这个步骤的执行期间可以接受增删改操作。所以，相对于整个**DDL**过程来说，锁的时间非常短。对业务来说，就可以认为是**Online**的。

需要补充说明的是，上述的这些重建方法都会扫描原表数据和构建临时文件。对于很大的表来说，这个操作是很消耗**IO**和**CPU**资源的。因此，如果是线上服务，你要很小心地控制操作时

间。如果想要比较安全的操作的话，我推荐你使用GitHub开源的gh-ost来做。

Online 和 inplace

说到Online，我还要再和你澄清一下它和另一个跟DDL有关的、容易混淆的概念inplace的区别。

你可能注意到了，在图3中，我们把表A中的数据导出来的存放位置叫作tmp_table。这是一个临时表，是在server层创建的。

在图4中，根据表A重建出来的数据是放在“tmp_file”里的，这个临时文件是InnoDB在内部创建出来的。整个DDL过程都在InnoDB内部完成。对于server层来说，没有把数据挪动到临时表，是一个“原地”操作，这就是“inplace”名称的来源。

所以，我现在问你，如果你有一个1TB的表，现在磁盘间是1.2TB，能不能做一个inplace的DDL呢？

答案是不能。因为，tmp_file也是要占用临时空间的。

我们重建表的这个语句alter table t engine=InnoDB，其实隐含的意思是：

```
alter table t engine=innodb,ALGORITHM=inplace;
```

跟inplace对应的就是拷贝表的方式了，用法是：

```
alter table t engine=innodb,ALGORITHM=copy;
```

当你使用ALGORITHM=copy的时候，表示的是强制拷贝表，对应的流程就是图3的操作过程。

但我这样说你可能会觉得，inplace跟Online是不是就是一个意思？

其实不是的，只是在重建表这个逻辑中刚好是这样而已。

比如，如果我要给InnoDB表的一个字段加全文索引，写法是：

```
alter table t add FULLTEXT(field_name);
```

这个过程是inplace的，但会阻塞增删改操作，是非Online的。

如果说这两个逻辑之间的关系是什么的话，可以概括为：

1. DDL过程如果是Online的，就一定是inplace的；

2. 反过来未必，也就是说`inplace`的DDL，有可能不是Online的。截止到MySQL 8.0，添加全文索引（`FULLTEXT index`）和空间索引（`SPATIAL index`）就属于这种情况。

最后，我们再延伸一下。

在第10篇文章[《MySQL为什么有时候会选错索引》](#)的评论区中，有同学问到使用`optimize table`、`analyze table`和`alter table`这三种方式重建表的区别。这里，我顺便再简单和你解释一下。

- 从MySQL 5.6版本开始，`alter table t engine = InnoDB`（也就是`recreate`）默认的就是上面图4的流程了；
- `analyze table t`其实不是重建表，只是对表的索引信息做重新统计，没有修改数据，这个过程中加了MDL读锁；
- `optimize table t`等于`recreate+analyze`。

小结

今天这篇文章，我和你讨论了数据库中收缩表空间的方法。

现在你已经知道了，如果要收缩一个表，只是`delete`掉表里面不用的数据的话，表文件的大小是不会变的，你还要通过`alter table`命令重建表，才能达到表文件变小的目的。我跟你介绍了重建表的两种实现方式，`Online DDL`的方式是可以考虑在业务低高峰期使用的，而MySQL 5.5及之前的版本，这个命令是会阻塞DML的，这个你需要特别小心。

最后，又到了我们的课后问题时间。

假设现在有人碰到了一个“想要收缩表空间，结果适得其反”的情况，看上去是这样的：

1. 一个表t文件大小为1TB；
2. 对这个表执行 `alter table t engine=InnoDB`；
3. 发现执行完成后，空间不仅没变小，还稍微大了一点儿，比如变成了1.01TB。

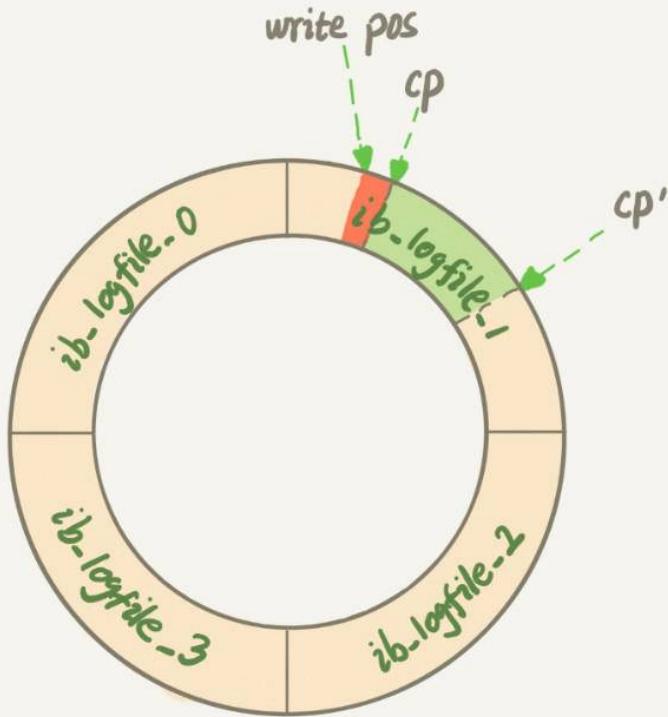
你觉得可能是什么原因呢？

你可以把你觉得可能的原因写在留言区里，我会在下一篇文章的末尾把大家描述的合理的原因都列出来，以后其他同学就不用掉到这样的坑里了。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

在上期文章最后，我留给你的问题是，如果一个高配的机器，`redo log`设置太小，会发生什么情况。

每次事务提交都要写redo log，如果设置太小，很快就会被写满，也就是下面这个图的状态，这个“环”将很快被写满，write pos一直追着CP。



这时候系统不得不停止所有更新，去推进checkpoint。

这时，你看到的现象就是磁盘压力很小，但是数据库出现间歇性的性能下跌。

评论区留言点赞板：

@某、人 给了一个形象的描述，而且提到了，在这种情况下，连change buffer的优化也失效了。因为checkpoint一直要往前推，这个操作就会触发merge操作，然后又进一步地触发刷脏页操作；

有几个同学提到了内存淘汰脏页，对应的redo log的操作，这个我们会在后面的文章中展开，大家可以先看一下 @melon 同学的描述了解一下；

@算不出流源 提到了“动态平衡”，其实只要出现了这种“平衡”，意味着本应该后台的操作，就已经影响了业务应用，属于有损失的平衡。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



14 | count(*)这么慢，我该怎么办？

2018-12-14 林晓斌



在开发系统的时候，你可能经常需要计算一个表的行数，比如一个交易系统的所有变更记录总数。这时候你可能会想，一条`select count(*) from t` 语句不就解决了吗？

但是，你会发现随着系统中记录数越来越多，这条语句执行得也会越来越慢。然后你可能就想了，MySQL怎么这么笨啊，记个总数，每次要查的时候直接读出来，不就好了吗。

那么今天，我们就来聊聊`count(*)`语句到底是怎样实现的，以及MySQL为什么会这么实现。然后，我会再和你说说，如果应用中有这种频繁变更并需要统计表行数的需求，业务设计上可以怎么做。

count(*)的实现方式

你首先要明确的是，在不同的MySQL引擎中，`count(*)`有不同的实现方式。

- MyISAM引擎把一个表的总行数存在了磁盘上，因此执行`count(*)`的时候会直接返回这个数，效率很高；
- 而InnoDB引擎就麻烦了，它执行`count(*)`的时候，需要把数据一行一行地从引擎里面读出来，然后累积计数。

这里需要注意的是，我们在这篇文章里讨论的是没有过滤条件的`count(*)`，如果加了`where`条件的话，MyISAM表也是不能返回得这么快的。

在前面的文章中，我们一起分析了为什么要使用InnoDB，因为不论是在事务支持、并发能力还是在数据安全方面，InnoDB都优于MyISAM。我猜你的表也一定是用了InnoDB引擎。这就是当你的记录数越来越多的时候，计算一个表的总行数会越来越慢的原因。

那为什么InnoDB不跟MyISAM一样，也把数字存起来呢？

这是因为即使是在同一个时刻的多个查询，由于多版本并发控制（MVCC）的原因，InnoDB表“应该返回多少行”也是不确定的。这里，我用一个算count(*)的例子来为你解释一下。

假设表t中现在有10000条记录，我们设计了三个用户并行的会话。

- 会话A先启动事务并查询一次表的总行数；
- 会话B启动事务，插入一行后记录后，查询表的总行数；
- 会话C先启动一个单独的语句，插入一行记录后，查询表的总行数。

我们假设从上到下是按照时间顺序执行的，同一行语句是在同一时刻执行的。

会话A	会话B	会话C
begin;		
select count(*) from t;		
		insert into t (插入一行);
	begin;	
	insert into t (插入一行);	
select count(*) from t; (返回10000)	select count(*) from t; (返回10002)	select count(*) from t; (返回10001)

图1 会话A、B、C的执行流程

你会看到，在最后一个时刻，三个会话A、B、C会同时查询表t的总行数，但拿到的结果却不同。

这和InnoDB的事务设计有关系，可重复读是它默认的隔离级别，在代码上就是通过多版本并发控制，也就是MVCC来实现的。每一行记录都要判断自己是否对这个会话可见，因此对于count(*)请求来说，InnoDB只好把数据一行一行地读出依次判断，可见的行才能够用于计算“基于这个查询”的表的总行数。

备注：如果你对MVCC记忆模糊了，可以再回顾下第3篇文章[《事务隔离：为什么你改了我还看不见？》](#)和第8篇文章[《事务到底是隔离的还是不隔离的？》](#)中的相关内容。

当然，现在这个看上去笨笨的MySQL，在执行count(*)操作的时候还是做了优化的。

你知道的，**InnoDB**是索引组织表，主键索引树的叶子节点是数据，而普通索引树的叶子节点是主键值。所以，普通索引树比主键索引树小很多。对于**count(*)**这样的操作，遍历哪个索引树得到的结果逻辑上都是一样的。因此，**MySQL**优化器会找到最小的那棵树来遍历。在保证逻辑正确的前提下，尽量减少扫描的数据量，是数据库系统设计的通用法则之一。

如果你用过**show table status**命令的话，就会发现这个命令的输出结果里面也有一个**TABLE_ROWS**用于显示这个表当前有多少行，这个命令执行挺快的，那这个**TABLE_ROWS**能代替**count(*)**吗？

你可能还记得在第10篇文章[《MySQL为什么有时候会选错索引？》](#)中我提到过，索引统计的值是通过采样来估算的。实际上，**TABLE_ROWS**就是从这个采样估算得来的，因此它也很不准。有多不准呢，官方文档说误差可能达到40%到50%。所以，**show table status**命令显示的行数也不能直接使用。

到这里我们小结一下：

- **MyISAM**表虽然**count(*)**很快，但是不支持事务；
- **show table status**命令虽然返回很快，但是不准确；
- **InnoDB**表直接**count(*)**会遍历全表，虽然结果准确，但会导致性能问题。

那么，回到文章开头的问题，如果你现在有一个页面经常要显示交易系统的操作记录总数，到底应该怎么办呢？答案是，我们只能自己计数。

接下来，我们讨论一下，看看自己计数有哪些方法，以及每种方法的优缺点有哪些。

这里，我先和你说一下这些方法的基本思路：你需要自己找一个地方，把操作记录表的行数存起来。

用缓存系统保存计数

对于更新很频繁的库来说，你可能会第一时间想到，用缓存系统来支持。

你可以用一个**Redis**服务来保存这个表的总行数。这个表每被插入一行**Redis**计数就加1，每被删除一行**Redis**计数就减1。这种方式下，读和更新操作都很快，但你再想一下这种方式存在什么问题吗？

没错，缓存系统可能会丢失更新。

Redis的数据不能永久地留在内存里，所以你会找一个地方把这个值定期地持久化存储起来。但即使这样，仍然可能丢失更新。试想如果刚刚在数据表中插入了一行，**Redis**中保存的值也加了1，然后**Redis**异常重启了，重启后你要从存储**redis**数据的地方把这个值读回来，而刚刚加1的这个计数操作却丢失了。

当然了，这还是有解的。比如，Redis异常重启以后，到数据库里面单独执行一次count(*)获取真实的行数，再把这个值写回到Redis里就可以了。异常重启毕竟不是经常出现的情况，这一次全表扫描的成本，还是可以接受的。

但实际上，将计数保存在缓存系统中的方式，还不只是丢失更新的问题。即使Redis正常工作，这个值还是逻辑上不精确的。

你可以设想一下有这么一个页面，要显示操作记录的总数，同时还要显示最近操作的100条记录。那么，这个页面的逻辑就需要先到Redis里面取出计数，再到数据表里面取数据记录。

我们是这么定义不精确的：

1. 一种是，查到的100行结果里面有最新插入记录，而Redis的计数里还没加1；
2. 另一种是，查到的100行结果里没有最新插入的记录，而Redis的计数里已经加了1。

这两种情况，都是逻辑不一致的。

我们一起来看看这个时序图。

时刻	会话A	会话B
T1		
T2	插入一行数据R；	
T3		读Redis计数； 查询最近100条记录；
T4	Redis计数加1；	

图2 会话A、B执行时序图

图2中，会话A是一个插入交易记录的逻辑，往数据表里插入一行R，然后Redis计数加1；会话B就是查询页面显示时需要的数据。

在图2的这个时序里，在T3时刻会话B来查询的时候，会显示出新插入的R这个记录，但是Redis的计数还没加1。这时候，就会出现我们说的数据不一致。

你一定会说，这是因为我们执行新增记录逻辑时候，是先写数据表，再改Redis计数。而读的时候是先读Redis，再读数据表，这个顺序是相反的。那么，如果保持顺序一样的话，是不是就没问题了？我们现在把会话A的更新顺序换一下，再看看执行结果。

时刻	会话A	会话B
T1		
T2	Redis 计数加1;	
T3		读Redis计数; 查询最近100条记录;
T4	插入一行数据R;	
T5		

图3 调整顺序后，会话A、B的执行时序图

你会发现，这时候反过来了，会话B在T3时刻查询的时候，Redis计数加了1了，但还查不到新插入的R这一行，也是数据不一致的情况。

在并发系统里面，我们是无法精确控制不同线程的执行时刻的，因为存在图中的这种操作序列，所以，我们说即使Redis正常工作，这个计数值还是逻辑上不精确的。

在数据库保存计数

根据上面的分析，用缓存系统保存计数有丢失数据和计数不精确的问题。那么，如果我们把这个计数直接放到数据库里单独的一张计数表C中，又会怎么样呢？

首先，这解决了崩溃丢失的问题，InnoDB是支持崩溃恢复不丢数据的。

备注：关于InnoDB的崩溃恢复，你可以再回顾一下第2篇文章[《日志系统：一条SQL更新语句是如何执行的？》](#)中的相关内容。

然后，我们再看看能不能解决计数不精确的问题。

你会说，这不一样吗？无非就是把图3中对Redis的操作，改成了对计数表C的操作。只要出现图3的这种执行序列，这个问题还是无解的吧？

这个问题还真不是无解的。

我们这篇文章要解决的问题，都是由于InnoDB要支持事务，从而导致InnoDB表不能把count(*)直接存起来，然后查询的时候直接返回形成的。

所谓以子之矛攻子之盾，现在我们就利用“事务”这个特性，把问题解决掉。

时刻	会话A	会话B
T1		
T2	begin; 表C中计数值加1;	
T3		begin; 读表C计数值; 查询最近100条记录; commit;
T4	插入一行数据R commit;	

图4 会话A、B的执行时序图

我们来看下现在的执行结果。虽然会话B的读操作仍然是在T3执行的，但是因为这时候更新事务还没有提交，所以计数值加1这个操作对会话B还不可见。

因此，会话B看到的结果里，查计数值和“最近100条记录”看到的结果，逻辑上就是一致的。

不同的**count**用法

在前面文章的评论区，有同学留言问到：在**select count(?) from t**这样的查询语句里面，**count(*)**、**count(主键id)**、**count(字段)**和**count(1)**等不同用法的性能，有哪些差别。今天谈到了**count(*)**的性能问题，我就借此机会和你详细说明一下这几种用法的性能差别。

需要注意的是，下面的讨论还是基于**InnoDB**引擎的。

这里，首先你要弄清楚**count()**的语义。**count()**是一个聚合函数，对于返回的结果集，一行行地判断，如果**count**函数的参数不是**NULL**，累计值就加1，否则不加。最后返回累计值。

所以，**count(*)**、**count(主键id)**和**count(1)**都表示返回满足条件的结果集的总行数；而**count(字段)**，则表示返回满足条件的数据行里面，参数“字段”不为**NULL**的总个数。

至于分析性能差别的时候，你可以记住这么几个原则：

1. **server**层要什么就给什么；
2. **InnoDB**只给必要的值；
3. 现在的优化器只优化了**count(*)**的语义为“取行数”，其他“显而易见”的优化并没有做。

这是什么意思呢？接下来，我们就一个个地来看看。

对于**count(主键id)**来说，InnoDB引擎会遍历整张表，把每一行的id值都取出来，返回给server层。server层拿到id后，判断是不可能为空的，就按行累加。

对于**count(1)**来说，InnoDB引擎遍历整张表，但不取值。server层对于返回的每一行，放一个数字“1”进去，判断是不可能为空的，按行累加。

单看这两个用法的差别的话，你能对比出来，**count(1)**执行得要比**count(主键id)**快。因为从引擎返回id会涉及到解析数据行，以及拷贝字段值的操作。

对于**count(字段)**来说：

1. 如果这个“字段”是定义为**not null**的话，一行行地从记录里面读出这个字段，判断不能为**null**，按行累加；
2. 如果这个“字段”定义允许为**null**，那么执行的时候，判断到有可能是**null**，还要把值取出来再判断一下，不是**null**才累加。

也就是前面的第一条原则，server层要什么字段，InnoDB就返回什么字段。

但是**count(*)**是例外，并不会把全部字段取出来，而是专门做了优化，不取值。**count(*)**肯定不是**null**，按行累加。

看到这里，你一定会说，优化器就不能自己判断一下吗，主键id肯定非空啊，为什么不能按照**count(*)**来处理，多么简单的优化啊。

当然，MySQL专门针对这个语句进行优化，也不是不可以。但是这种需要专门优化的情况太多了，而且MySQL已经优化过**count(*)**了，你直接使用这种用法就可以了。

所以结论是：按照效率排序的话，**count(字段) < count(主键id) < count(1) ≈ count(*)**，所以我建议你，尽量使用**count(*)**。

小结

今天，我和你聊了聊MySQL中获得表行数的两种方法。我们提到了在不同引擎中**count(*)**的实现方式是不一样的，也分析了用缓存系统来存储计数值存在的问题。

其实，把计数放在Redis里面，不能够保证计数和MySQL表里的数据精确一致的原因，是这两个不同的存储构成的系统，不支持分布式事务，无法拿到精确一致的视图。而把计数值也放在MySQL中，就解决了一致性视图的问题。

InnoDB引擎支持事务，我们利用好事务的原子性和隔离性，就可以简化在业务开发时的逻辑。这也是InnoDB引擎备受青睐的原因之一。

最后，又到了今天的思考题时间了。

在刚刚讨论的方案中，我们用了事务来确保计数准确。由于事务可以保证中间结果不被别的事务读到，因此修改计数值和插入新记录的顺序是不影响逻辑结果的。但是，从并发系统性能的角度考虑，你觉得在这个事务序列里，应该先插入操作记录，还是应该先更新计数表呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾给出我的参考答案。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期我给你留的问题是，什么时候使用`alter table t engine=InnoDB`会让一个表占用的空间反而变大。

在这篇文章的评论区里面，大家都提到了一个点，就是这个表，本身就已经没有空洞的了，比如说刚刚做过一次重建表操作。

在`DDL`期间，如果刚好有外部的`DML`在执行，这期间可能会引入一些新的空洞。

@飞翔 提到了一个更深刻的机制，是我们在文章中没说的。在重建表的时候，`InnoDB`不会把整张表占满，每个页留了`1/16`给后续的更新用。也就是说，其实重建表之后不是“最”紧凑的。

假如是这么一个过程：

1. 将表`t`重建一次；
2. 插入一部分数据，但是插入的这些数据，用掉了一部分的预留空间；
3. 这种情况下，再重建一次表`t`，就可能会出现问题中的现象。

评论区留言点赞板：

@W_T 等同学提到了数据表本身紧凑的情况；

@undefined 提了一个好问题，@帆帆帆帆帆帆帆帆同学回答了这个问题；

@陈飞 @郜 @wang chen wen 都提了很不错的问题，大家可以去看看。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

15 | 答疑文章（一）：日志和索引相关问题

2018-12-17 林晓斌



在今天这篇答疑文章更新前，MySQL实战这个专栏已经更新了14篇。在这些文章中，大家在评论区留下了很多高质量的留言。现在，每篇文章的评论区都有热心的同学帮忙总结文章知识点，也有不少同学提出了很多高质量的问题，更有一些同学帮忙解答其他同学提出的问题。

在浏览这些留言并回复的过程中，我倍受鼓舞，也尽我所知地帮助你解决问题、和你讨论。可以说，你们的留言活跃了整个专栏的氛围、提升了整个专栏的质量，谢谢你们。

评论区的大多数留言我都直接回复了，对于需要展开说明的问题，我都拿出小本子记了下来。这些被记下来的问题，就是我们今天这篇答疑文章的素材了。

到目前为止，我已经收集了47个问题，很难通过今天这一篇文章全部展开。所以，我就先从中找了几个联系非常紧密的问题，串了起来，希望可以帮你解决关于日志和索引的一些疑惑。而其他问题，我们就留着后面慢慢展开吧。

日志相关问题

我在第2篇文章[《日志系统：一条SQL更新语句是如何执行的？》](#)中，和你讲到binlog（归档日志）和redo log（重做日志）配合崩溃恢复的时候，用的是反证法，说明了如果没有两阶段提交，会导致MySQL出现主备数据不一致等问题。

在这篇文章下面，很多同学在问，在两阶段提交的不同瞬间，MySQL如果发生异常重启，是怎么保证数据完整性的？

现在，我们就从这个问题开始吧。

我再放一次两阶段提交的图，方便你学习下面的内容。

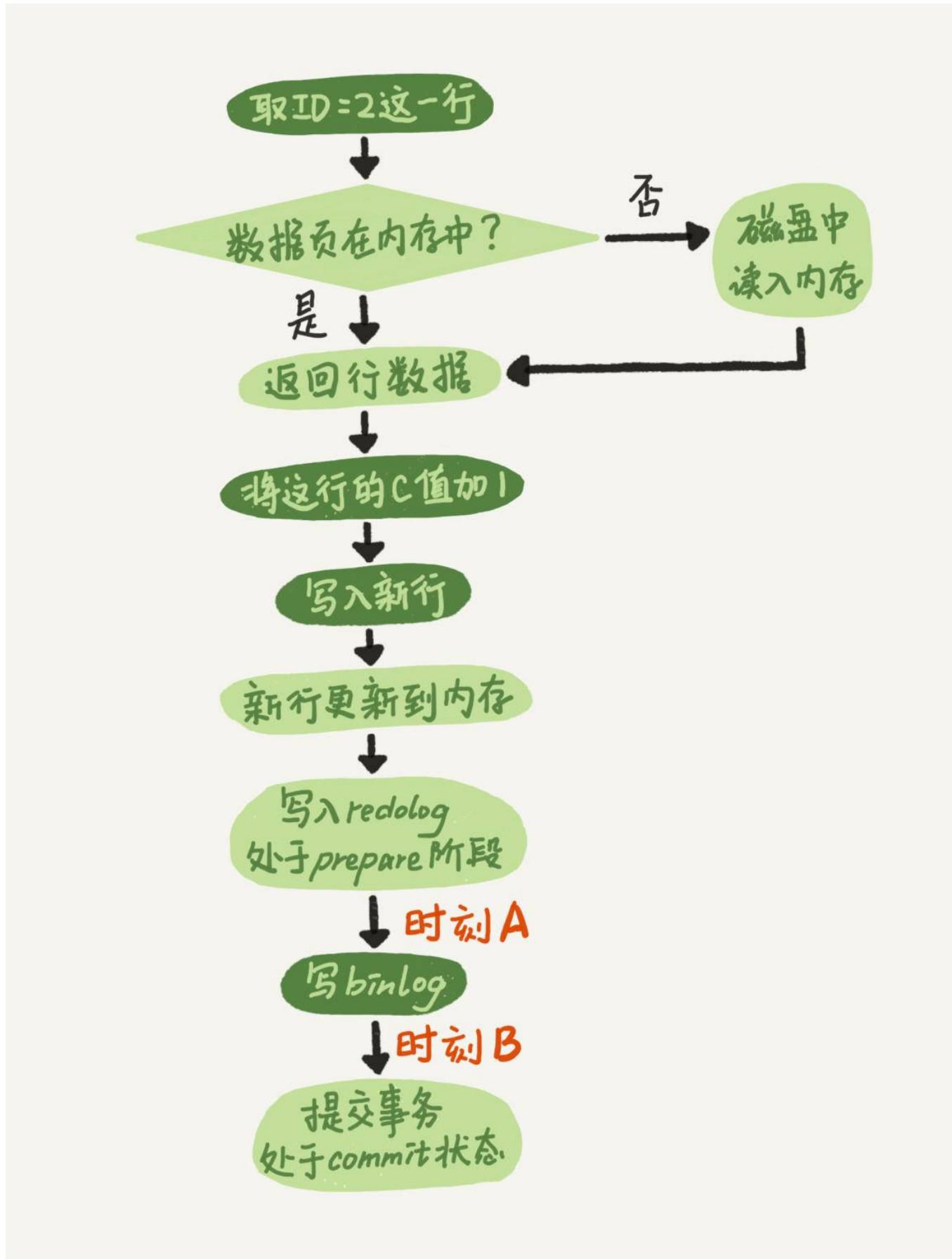


图1 两阶段提交示意图

这里，我要先和你解释一个误会式的问题。有同学在评论区问到，这个图不是一个update语句的执行流程吗，怎么还会调用commit语句？

他产生这个疑问的原因，是把两个“**commit**”的概念混淆了：

- 他说的“**commit**语句”，是指MySQL语法中，用于提交一个事务的命令。一般跟begin/start transaction配对使用。
- 而我们图中用到的这个“**commit**步骤”，指的是事务提交过程中的一个小步骤，也是最后一步。当这个步骤执行完成后，这个事务就提交完成了。
- “**commit**语句”执行的时候，会包含“**commit**步骤”。

而我们这个例子里面，没有显式地开启事务，因此这个update语句自己就是一个事务，在执行完成后提交事务时，就会用到这个“**commit**步骤”。

接下来，我们就一起分析一下在两阶段提交的不同时刻，MySQL异常重启会出现什么现象。

如果在图中时刻A的地方，也就是写入redo log 处于prepare阶段之后、写binlog之前，发生了崩溃（crash），由于此时binlog还没写，redo log也还没提交，所以崩溃恢复的时候，这个事务会回滚。这时候，binlog还没写，所以也不会传到备库。到这里，大家都可以理解。

大家出现问题的地方，主要集中在时刻B，也就是binlog写完，redo log还没**commit**前发生crash，那崩溃恢复的时候MySQL会怎么处理？

我们先来看一下崩溃恢复时的判断规则。

1. 如果redo log里面的事务是完整的，也就是已经有了**commit**标识，则直接提交；
2. 如果redo log里面的事务只有完整的**prepare**，则判断对应的事务binlog是否存在并完整：
 - a. 如果是，则提交事务；
 - b. 否则，回滚事务。

这里，时刻B发生crash对应的就是2(a)的情况，崩溃恢复过程中事务会被提交。

现在，我们继续延展一下这个问题。

追问1：MySQL怎么知道binlog是完整的？

回答：一个事务的binlog是有完整格式的：

- statement格式的binlog，最后会有**COMMIT**；
- row格式的binlog，最后会有一个**XID event**。

另外，在MySQL 5.6.2版本以后，还引入了**binlog-checksum**参数，用来验证binlog内容的正确性。对于binlog日志由于磁盘原因，可能会在日志中间出错的情况，MySQL可以通过校验

`checksum`的结果来发现。所以，MySQL还是有办法验证事务binlog的完整性的。

追问2：redo log 和 binlog是怎么关联起来的？

回答：它们有一个共同的数据字段，叫XID。崩溃恢复的时候，会按顺序扫描redo log：

- 如果碰到既有prepare、又有commit的redo log，就直接提交；
- 如果碰到只有parepare、而没有commit的redo log，就拿着XID去binlog找对应的事务。

追问3：处于prepare阶段的redo log加上完整binlog，重启就能恢复，MySQL为什么要这么设计？

回答：其实，这个问题还是跟我们在反证法中说到的数据与备份的一致性有关。在时刻B，也就是binlog写完以后MySQL发生崩溃，这时候binlog已经写入了，之后就会被从库（或者用这个binlog恢复出来的库）使用。

所以，在主库上也要提交这个事务。采用这个策略，主库和备库的数据就保证了一致性。

追问4：如果这样的话，为什么还要两阶段提交呢？干脆先redo log写完，再写binlog。崩溃恢复的时候，必须得两个日志都完整才可以。是不是一样的逻辑？

回答：其实，两阶段提交是经典的分布式系统问题，并不是MySQL独有的。

如果必须要举一个场景，来说明这么做的必要性的话，那就是事务的持久性问题。

对于InnoDB引擎来说，如果redo log提交完成了，事务就不能回滚（如果这还允许回滚，就可能覆盖掉别的事务的更新）。而如果redo log直接提交，然后binlog写入的时候失败，InnoDB又回滚不了，数据和binlog日志又不一致了。

两阶段提交就是为了给所有人一个机会，当每个人都说“我ok”的时候，再一起提交。

追问5：不引入两个日志，也就没有两阶段提交的必要了。只用binlog来支持崩溃恢复，又能支持归档，不就可以了？

回答：这位同学的意思是，只保留binlog，然后可以把提交流程改成这样：...->“数据更新到内存”->“写 binlog”->“提交事务”，是不是也可以提供崩溃恢复的能力？

答案是不可以。

如果说历史原因的话，那就是InnoDB并不是MySQL的原生存储引擎。MySQL的原生引擎是MyISAM，设计之初就没有支持崩溃恢复。

InnoDB在作为MySQL的插件加入MySQL引擎家族之前，就已经是一个提供了崩溃恢复和事务支持的引擎了。

InnoDB接入了MySQL后，发现既然binlog没有崩溃恢复的能力，那就用InnoDB原有的redo log

好了。

而如果说实现上的原因的话，就有很多了。就按照问题中说的，只用**binlog**来实现崩溃恢复的流程，我画了一张示意图，这里就没有**redo log**了。

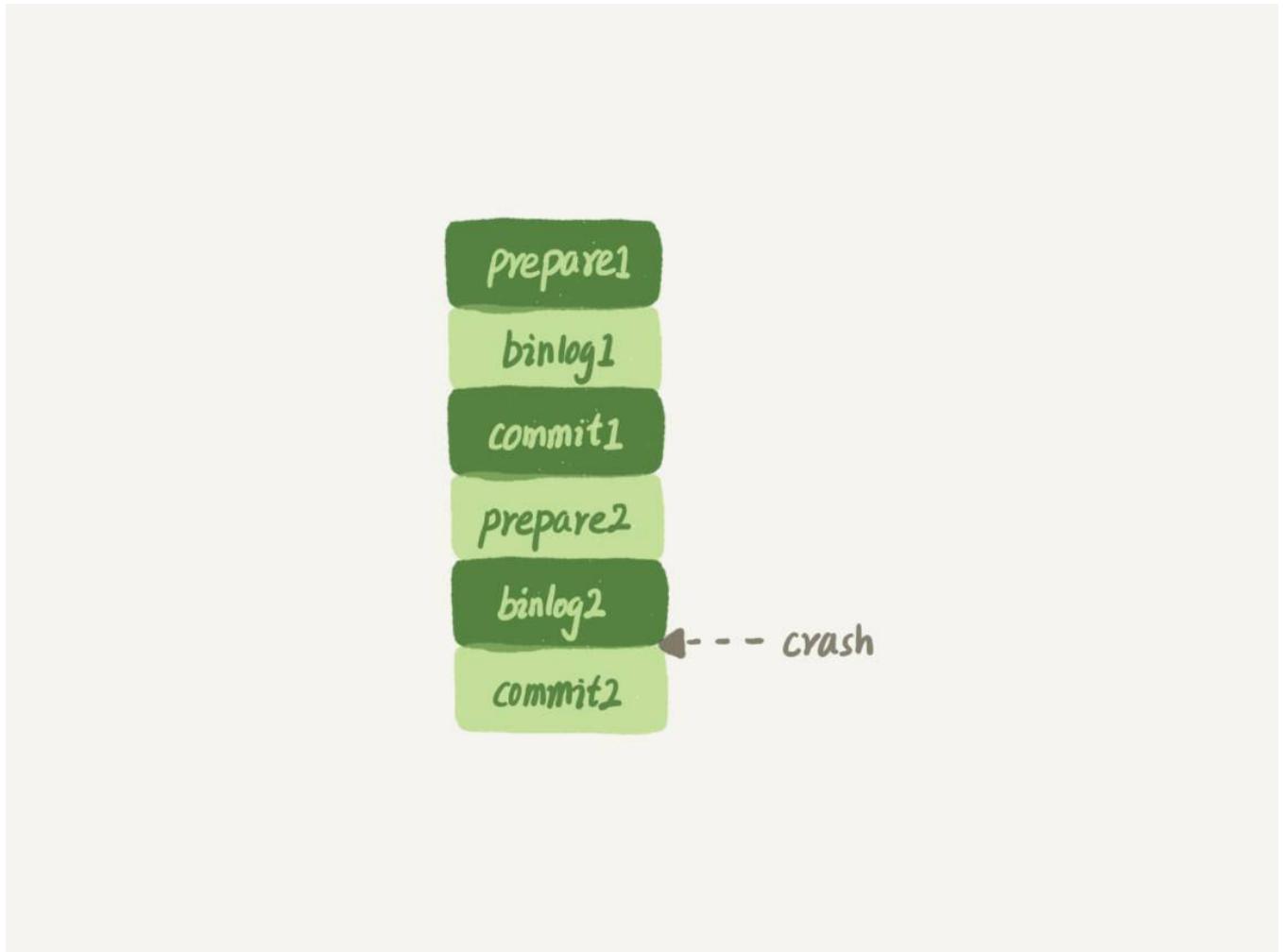


图2 只用**binlog**支持崩溃恢复

这样的流程下，**binlog**还是不能支持崩溃恢复的。我说一个不支持的点吧：**binlog**没有能力恢复“数据页”。

如果在图中标的位置，也就是**binlog2**写完了，但是整个事务还没有**commit**的时候，MySQL发生了**crash**。

重启后，引擎内部事务2会回滚，然后应用**binlog2**可以补回来；但是对于事务1来说，系统已经认为提交完成了，不会再应用一次**binlog1**。

但是，InnoDB引擎使用的是WAL技术，执行事务的时候，写完内存和日志，事务就算完成了。如果之后崩溃，要依赖于日志来恢复数据页。

也就是说在图中这个位置发生崩溃的话，事务1也是可能丢失了的，而且是数据页级的丢失。此时，**binlog**里面并没有记录数据页的更新细节，是补不回来的。

你如果说，那我优化一下**binlog**的内容，让它来记录数据页的更改可以吗？但，这其实就是一个**redo log**。

所以，至少现在的**binlog**能力，还不能支持崩溃恢复。

追问6：那能不能反过来，只用**redo log**，不要**binlog**？

回答：如果只从崩溃恢复的角度来讲是可以的。你可以把**binlog**关掉，这样就没有两阶段提交了，但系统依然是**crash-safe**的。

但是，如果你了解一下业界各个公司的使用场景的话，就会发现在正式的生产库上，**binlog**都是开着的。因为**binlog**有着**redo log**无法替代的功能。

一个是归档。**redo log**是循环写，写到末尾是要回到开头继续写的。这样历史日志没法保留，**redo log**也就起不到归档的作用。

一个就是MySQL系统依赖于**binlog**。**binlog**作为MySQL一开始就有功能，被用在了很多地方。其中，MySQL系统高可用的基础，就是**binlog**复制。

还有很多公司有异构系统（比如一些数据分析系统），这些系统就靠消费MySQL的**binlog**来更新自己的数据。关掉**binlog**的话，这些下游系统就没法输入了。

总之，由于现在包括MySQL高可用在内的很多系统机制都依赖于**binlog**，所以“鸠占鹊巢”**redo log**还做不到。你看，发展生态是多么重要。

追问7：**redo log**一般设置多大？

回答：**redo log**太小的话，会导致很快就被写满，然后不得不强行刷**redo log**，这样WAL机制的能力就发挥不出来了。

所以，如果是现在常见的几个TB的磁盘的话，就不要太小气了，直接将**redo log**设置为4个文件、每个文件1GB吧。

追问8：正常运行中的实例，数据写入后的最终落盘，是从**redo log**更新过来的还是从**buffer pool**更新过来的呢？

回答：这个问题其实问得非常好。这里涉及到了，“**redo log**里面到底是什么”的问题。

实际上，**redo log**并没有记录数据页的完整数据，所以它并没有能力自己去更新磁盘数据页，也就不存在“数据最终落盘，是由**redo log**更新过去”的情况。

1. 如果是正常运行的实例的话，数据页被修改以后，跟磁盘的数据页不一致，称为脏页。最终数据落盘，就是把内存中的数据页写盘。这个过程，甚至与**redo log**毫无关系。
2. 在崩溃恢复场景中，InnoDB如果判断到一个数据页可能在崩溃恢复的时候丢失了更新，就

会将它读到内存，然后让**redo log**更新内存内容。更新完成后，内存页变成脏页，就回到了第一种情况的状态。

追问9：**redo log buffer**是什么？是先修改内存，还是先写**redo log**文件？

回答：这两个问题可以一起回答。

在一个事务的更新过程中，日志是要写多次的。比如下面这个事务：

```
begin;
insert into t1 ...
insert into t2 ...
commit;
```

这个事务要往两个表中插入记录，插入数据的过程中，生成的日志都得先保存起来，但又不能在还没**commit**的时候就直接写到**redo log**文件里。

所以，**redo log buffer**就是一块内存，用来先存**redo**日志的。也就是说，在执行第一个**insert**的时候，数据的内存被修改了，**redo log buffer**也写入了日志。

但是，真正把日志写到**redo log**文件（文件名是 **ib_logfile+数字**），是在执行**commit**语句的时候做的。

（这里说的是事务执行过程中不会“主动去刷盘”，以减少不必要的IO消耗。但是可能会出现“被动写入磁盘”，比如内存不够、其他事务提交等情况。这个问题我们会在后面第22篇文章《MySQL有哪些“饮鸩止渴”的提高性能的方法？」中再详细展开）。

单独执行一个更新语句的时候，**InnoDB**会自己启动一个事务，在语句执行完成的时候提交。过程跟上面是一样的，只不过是“压缩”到了一个语句里面完成。

以上这些问题，就是把大家提过的关于**redo log**和**binlog**的问题串起来，做的一次集中回答。如果你还有问题，可以在评论区继续留言补充。

业务设计问题

接下来，我再和你分享@ithunter 同学在第8篇文章[《事务到底是隔离的还是不隔离的？」](#)的评论区提到的跟索引相关的一个问题。我觉得这个问题挺有趣、也挺实用的，其他同学也可能会碰上这样的场景，在这里解答和分享一下。

问题是这样的（我文字上稍微做了点修改，方便大家理解）：

业务上有这样的需求，A、B两个用户，如果互相关注，则成为好友。设计上是有两张表，一个是**like**表，一个是**friend**表，**like**表有**user_id**、**liker_id**两个字段，我设置为复合唯一索引即

`uk_user_id_liker_id`。语句执行逻辑是这样的：

以A关注B为例：

第一步，先查询对方有没有关注自己（B有没有关注A）

```
select * from like where user_id = B and liker_id = A;
```

如果有，则成为好友

```
insert into friend;
```

没有，则只是单向关注关系

```
insert into like;
```

但是如果A、B同时关注对方，会出现不会成为好友的情况。因为上面第1步，双方都没关注对方。第1步即使使用了排他锁也不行，因为记录不存在，行锁无法生效。请问这种情况，在MySQL锁层面有没有办法处理？

首先，我要先赞一下这样的提问方式。虽然极客时间现在的评论区还不能追加评论，但如果大家能够一次留言就把问题讲清楚的话，其实影响也不大。所以，我希望你在留言提问的时候，也能借鉴这种方式。

接下来，我把@ithunter同学说的表模拟出来，方便我们讨论。

```
CREATE TABLE `like` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `user_id` int(11) NOT NULL,
    `liker_id` int(11) NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `uk_user_id_liker_id` (`user_id`, `liker_id`)
) ENGINE=InnoDB;
```

```
CREATE TABLE `friend` (
    id` int(11) NOT NULL AUTO_INCREMENT,
    `friend_1_id` int(11) NOT NULL,
    `friend_2_id` int(11) NOT NULL,
    UNIQUE KEY `uk_friend` (`friend_1_id`, `friend_2_id`)
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

虽然这个题干中，并没有说到**friend**表的索引结构。但我猜测**friend_1_id**和**friend_2_id**也有索

引，为便于描述，我给加上唯一索引。

顺便说明一下，“like”是关键字，我一般不建议使用关键字作为库名、表名、字段名或索引名。

我把他的疑问翻译一下，在并发场景下，同时有两个人，设置为关注对方，就可能导致无法成功加为朋友关系。

现在，我用你已经熟悉的时刻顺序表的形式，把这两个事务的执行语句列出来：

session 1 (操作逻辑：A喜欢B)	session 2 (操作逻辑：B喜欢A)
begin; select * from `like` where user_id = B and liker_id = A; (返回空)	
	begin; select * from `like` where user_id = A and liker_id = B; (返回空)
	insert into `like` (user_id, liker_id) values(B, A);
insert into `like` (user_id, liker_id) values(A, B);	
commit;	
	commit;

图3 并发“喜欢”逻辑操作顺序

由于一开始A和B之间没有关注关系，所以两个事务里面的select语句查出来的结果都是空。

因此，session 1的逻辑就是“既然B没有关注A，那就只插入一个单向关注关系”。session 2也同样是这个逻辑。

这个结果对业务来说就是bug了。因为在业务设定里面，这两个逻辑都执行完成以后，是应该在friend表里面插入一行记录的。

如提问里面说的，“第1步即使使用了排他锁也不行，因为记录不存在，行锁无法生效”。不过，我想到了另外一个方法，来解决这个问题。

首先，要给“like”表增加一个字段，比如叫作 `relation_ship`，并设为整型，取值1、2、3。

值是1的时候，表示`user_id`关注`liker_id`；

值是2的时候，表示`liker_id`关注`user_id`；

值是3的时候，表示互相关注。

然后，当A关注B的时候，逻辑改成如下所示的样子：

应用代码里面，比较A和B的大小，如果A<B，就执行下面的逻辑

```
mysql> begin; /*启动事务*/  
insert into `like`(user_id, liker_id, relation_ship) values(A, B, 1) on duplicate key update r  
select relation_ship from `like` where user_id=A and liker_id=B;  
/*代码中判断返回的 relation_ship,  
如果是1，事务结束，执行 commit  
如果是3，则执行下面这两个语句：  
*/  
insert ignore into friend(friend_1_id, friend_2_id) values(A,B);  
commit;
```

如果A>B，则执行下面的逻辑

```
mysql> begin; /*启动事务*/  
insert into `like`(user_id, liker_id, relation_ship) values(B, A, 2) on duplicate key update r  
select relation_ship from `like` where user_id=B and liker_id=A;  
/*代码中判断返回的 relation_ship,  
如果是2，事务结束，执行 commit  
如果是3，则执行下面这两个语句：  
*/  
insert ignore into friend(friend_1_id, friend_2_id) values(B,A);  
commit;
```

这个设计里，让“like”表里的数据保证user_id < liker_id，这样不论是A关注B，还是B关注A，在操作“like”表的时候，如果反向的关系已经存在，就会出现行锁冲突。

然后，`insert ...on duplicate`语句，确保了在事务内部，执行了这个SQL语句后，就强行占住了这个行锁，之后的`select`判断`relation_ship`这个逻辑时就确保了是在行锁保护下的读操作。

操作符“|”是按位或，连同最后一句`insert`语句里的`ignore`，是为了保证重复调用时的幂等性。

这样，即使在双方“同时”执行关注操作，最终数据库里的结果，也是`like`表里面有一条关于A和B的记录，而且`relation_ship`的值是3，并且`friend`表里面也有了A和B的这条记录。

不知道你会不会吐槽：之前明明还说尽量不要使用唯一索引，结果这个例子一上来我就创建了两个。这里我要再和你说明一下，之前文章我们讨论的，是在“业务开发保证不会插入重复记录”的情况下，着重要解决性能问题的时候，才建议尽量使用普通索引。

而像这个例子里，按照这个设计，业务根本就是保证“我一定会插入重复数据，数据库一定要要有唯一性约束”，这时就没啥好说的了，唯一索引建起来吧。

小结

这是专栏的第一篇答疑文章。

我针对前14篇文章，大家在评论区中的留言，从中摘取了关于日志和索引的相关问题，串成了今天这篇文章。这里我也要再和你说一声，有些我答应在答疑文章中进行扩展的话题，今天这篇文章没来得及扩展，后续我会再找机会为你解答。所以，篇幅所限，评论区见吧。

最后，虽然这篇是答疑文章，但课后问题还是要有的。

我们创建了一个简单的表t，并插入一行，然后对这一行做修改。

```
mysql> CREATE TABLE `t` (
    `id` int(11) NOT NULL primary key auto_increment,
    `a` int(11) DEFAULT NULL
) ENGINE=InnoDB;
insert into t values(1,2);
```

这时候，表t里有唯一的一行数据(1,2)。假设，我现在要执行：

```
mysql> update t set a=2 where id=1;
```

你会看到这样的结果：

```
mysql> update t set a=2 where id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0
```

结果显示，匹配(rows matched)了一行，修改(Changed)了0行。

仅从现象上看，MySQL内部在处理这个命令的时候，可以有以下三种选择：

1. 更新都是先读后写的，MySQL读出数据，发现a的值本来就是2，不更新，直接返回，执行

结束；

2. MySQL调用了InnoDB引擎提供的“修改为(1,2)”这个接口，但是引擎发现值与原来相同，不更新，直接返回；
3. InnoDB认真执行了“把这个值修改成(1,2)”这个操作，该加锁的加锁，该更新的更新。

你觉得实际情况会是以上哪种呢？你可否用构造实验的方式，来证明你的结论？进一步地，可以思考一下，MySQL为什么要选择这种策略呢？

你可以把你的验证方法和思考写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，用一个计数表记录一个业务表的总行数，在往业务表插入数据的时候，需要给计数值加1。

逻辑实现上是启动一个事务，执行两个语句：

1. `insert into` 数据表；
2. `update` 计数表，计数值加1。

从系统并发能力的角度考虑，怎么安排这两个语句的顺序。

这里，我直接复制 @阿建 的回答过来供你参考：

并发系统性能的角度考虑，应该先插入操作记录，再更新计数表。

知识点在 [《行锁功过：怎么减少行锁对性能的影响？》](#)

因为更新计数表涉及到行锁的竞争，先插入再更新能最大程度地减少事务之间的锁等待，提升并发度。

评论区有同学说，应该把`update`计数表放后面，因为这个计数表可能保存了多个业务表的计数值。如果把`update`计数表放到事务的第一个语句，多个业务表同时插入数据的话，等待时间会更长。

这个答案的结论是对的，但是理解不太正确。即使我们用一个计数表记录多个业务表的行数，也肯定会给表名字段加唯一索引。类似于下面这样的表结构：

```
CREATE TABLE `rows_stat` (
  `table_name` varchar(64) NOT NULL,
  `row_count` int(10) unsigned NOT NULL,
  PRIMARY KEY (`table_name`)
) ENGINE=InnoDB;
```

在更新计数表的时候，一定会传入`where table_name=$table_name`，使用主键索引，更新加行锁只会锁在一行上。

而在不同业务表插入数据，是更新不同的行，不会有行锁。

评论区留言点赞板：

@北天魔狼、@斜面镜子 Bil 和 @Bin 等同学，都给出了正确答案；
@果然如此 同学提了一个好问题，虽然引入事务，避免看到“业务上还没提交的更新”，但是 Redis 的计数被提前看到了。核心原因还是两个系统，不支持一致性视图；
@帆帆帆帆帆帆帆帆同学的问题提醒了大家，`count(id)`也是可以走普通索引得到的。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Lin Xiaobin, a man with glasses and a black shirt, standing with his arms crossed. To his left is the course title "MySQL 实战 45 讲" and a subtitle "从原理到实战，丁奇带你搞懂 MySQL". Below the title is the instructor's name "林晓斌" and his alias "网名丁奇 前阿里资深技术专家". At the bottom, there is a call-to-action: "新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。"

16 | “order by”是怎么工作的？

2018-12-19 林晓斌



在你开发应用的时候，一定会经常碰到需要根据指定的字段排序来显示结果的需求。还是以我们前面举例用过的市民表为例，假设你要查询城市是“杭州”的所有名字，并且按照姓名排序返回前1000个人的姓名、年龄。

假设这个表的部分定义是这样的：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `city` varchar(16) NOT NULL,
  `name` varchar(16) NOT NULL,
  `age` int(11) NOT NULL,
  `addr` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `city` (`city`)
) ENGINE=InnoDB;
```

这时，你的SQL语句可以这么写：

```
select city,name,age from t where city='杭州' order by name limit 1000 ;
```

这个语句看上去逻辑很清晰，但是你了解它的执行流程吗？今天，我就和你聊聊这个语句是怎么执行的，以及有什么参数会影响执行的行为。

全字段排序

前面我们介绍过索引，所以你现在就很清楚了，为避免全表扫描，我们需要在city字段加上索引。

在city字段上创建索引之后，我们用explain命令来看看这个语句的执行情况。

```
mysql> explain select city, name, age from T where city='杭州' order by name limit 1000;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | partitions | type | possible_keys | key | key_len | ref  | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE     | T      | NULL       | ref  | city          | city | 51    | const | 4000 | 100.00 | Using index condition; Using filesort |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图1 使用explain命令查看语句的执行情况

Extra这个字段中的“Using filesort”表示的就是需要排序，MySQL会给每个线程分配一块内存用于排序，称为sort_buffer。

为了说明这个SQL查询语句的执行过程，我们先来看一下city这个索引的示意图。

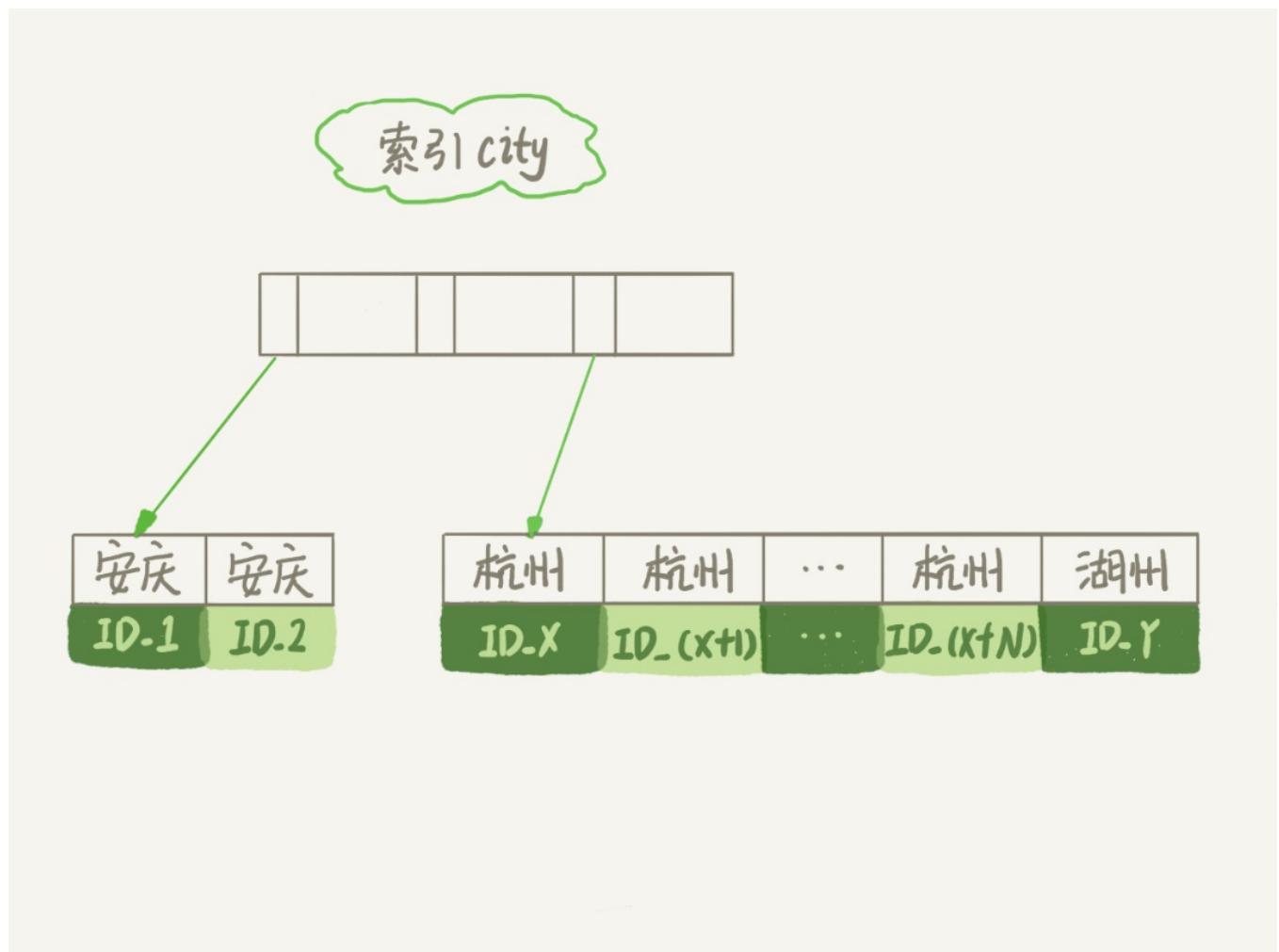


图2 city字段的索引示意图

从图中可以看到，满足city='杭州'条件的行，是从ID_X到ID_(X+N)的这些记录。

通常情况下，这个语句执行流程如下所示：

1. 初始化sort_buffer，确定放入name、city、age这三个字段；
2. 从索引city找到第一个满足city='杭州'条件的主键id，也就是图中的ID_X；
3. 到主键id索引取出整行，取name、city、age三个字段的值，存入sort_buffer中；
4. 从索引city取下一个记录的主键id；
5. 重复步骤3、4直到city的值不满足查询条件为止，对应的主键id也就是图中的ID_Y；
6. 对sort_buffer中的数据按照字段name做快速排序；
7. 按照排序结果取前1000行返回给客户端。

我们暂且把这个排序过程，称为全字段排序，执行流程的示意图如下所示，下一篇文章中我们还会用到这个排序。

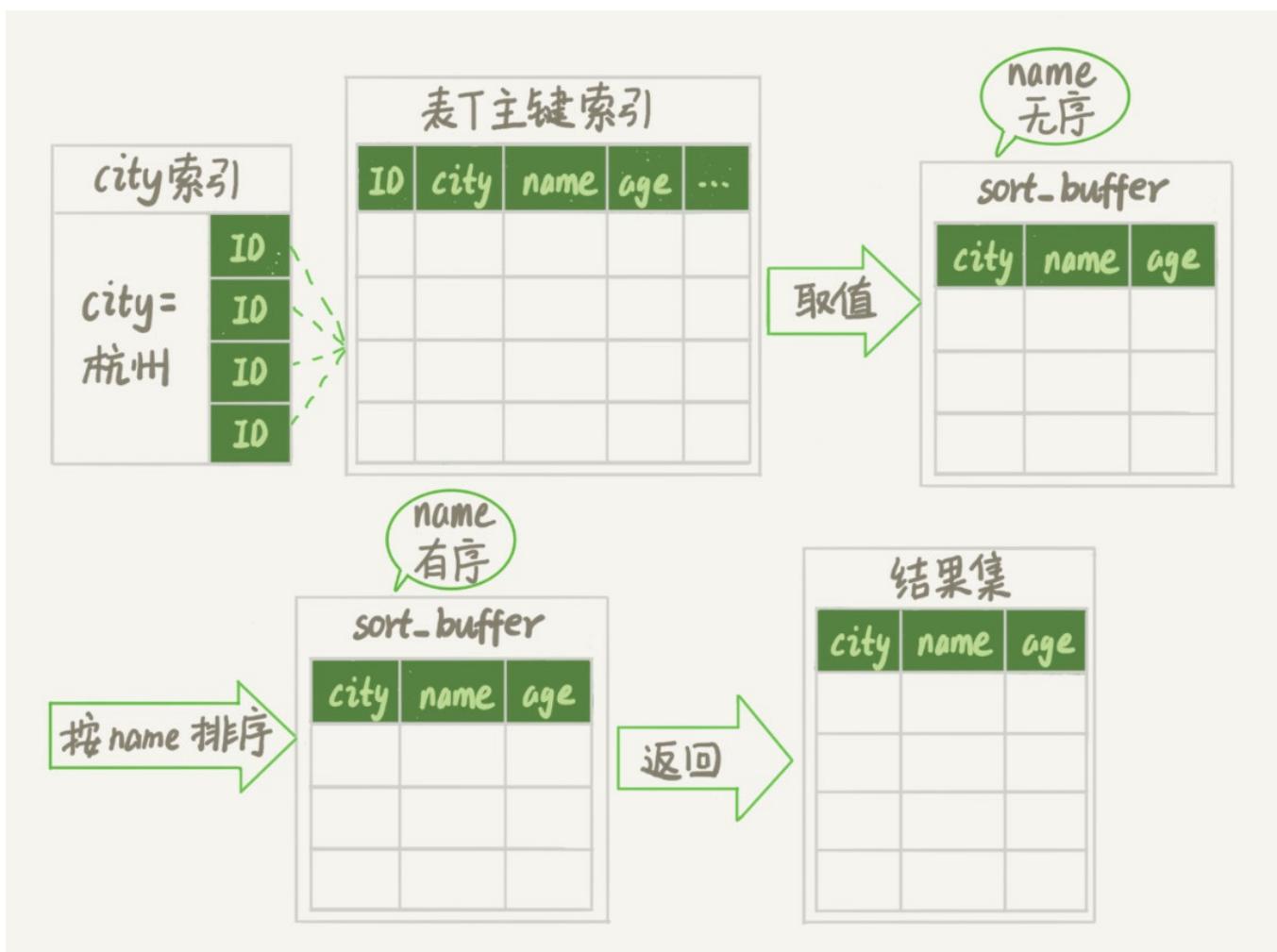


图3 全字段排序

图中“按name排序”这个动作，可能在内存中完成，也可能需要使用外部排序，这取决于排序所需的内存和参数sort_buffer_size。

sort_buffer_size，就是MySQL为排序开辟的内存（sort_buffer）的大小。如果要排序的数据量小于sort_buffer_size，排序就在内存中完成。但如果排序数据量太大，内存放不下，则不得不利用磁盘临时文件辅助排序。

你可以用下面介绍的方法，来确定一个排序语句是否使用了临时文件。

```
/* 打开optimizer_trace，只对本线程有效 */
SET optimizer_trace='enabled=on';

/* @a保存innodb_rows_read的初始值 */
select VARIABLE_VALUE into @a from performance_schema.session_status where variable_name = 'innodb_rows_read';

/* 执行语句 */
select city, name, age from t where city='杭州' order by name limit 1000;

/* 查看 OPTIMIZER_TRACE 输出 */
SELECT * FROM `information_schema`.`OPTIMIZER_TRACE`\\G

/* @b保存innodb_rows_read的当前值 */
select VARIABLE_VALUE into @b from performance_schema.session_status where variable_name = 'innodb_rows_read';

/* 计算innodb_rows_read差值 */
select @b-@a;
```

这个方法是通过查看 OPTIMIZER_TRACE 的结果来确认的，你可以从 number_of_tmp_files 中看到是否使用了临时文件。

```
"filesort_execution": [
],
"filesort_summary": {
    "rows": 4000,
    "examined_rows": 4000,
    "number_of_tmp_files": 12,
    "sort_buffer_size": 32684,
    "sort_mode": "<sort_key, packed_additional_fields>"
}
```

图4 全排序的OPTIMIZER_TRACE部分结果

`number_of_tmp_files`表示的是，排序过程中使用的临时文件数。你一定奇怪，为什么需要12个文件？内存放不下时，就需要使用外部排序，外部排序一般使用归并排序算法。可以这么简单理解，MySQL将需要排序的数据分成12份，每一份单独排序后存在这些临时文件中。然后把这12个有序文件再合并成一个有序的大文件。

如果`sort_buffer_size`超过了需要排序的数据量的大小，`number_of_tmp_files`就是0，表示排序可以直接在内存中完成。

否则就需要放在临时文件中排序。`sort_buffer_size`越小，需要分成的份数越多，`number_of_tmp_files`的值就越大。

接下来，我再和你解释一下图4中其他两个值的意思。

我们的示例表中有4000条满足`city='杭州'`的记录，所以你可以看到`examined_rows=4000`，表示参与排序的行数是4000行。

`sort_mode`里面的`packed_additional_fields`的意思是，排序过程对字符串做了“紧凑”处理。即使`name`字段的定义是`varchar(16)`，在排序过程中还是要按照实际长度来分配空间的。

同时，最后一个查询语句`select @b-@a`的返回结果是4000，表示整个执行过程只扫描了4000行。

这里需要注意的是，为了避免对结论造成干扰，我把`internal_tmp_disk_storage_engine`设置成MyISAM。否则，`select @b-@a`的结果会显示为4001。

这是因为查询OPTIMIZER_TRACE这个表时，需要用到临时表，而`internal_tmp_disk_storage_engine`的默认值是InnoDB。如果使用的是InnoDB引擎的话，把数据从临时表取出来的时候，会让`Innodb_rows_read`的值加1。

rowid排序

在上面这个算法过程里面，只对原表的数据读了一遍，剩下的操作都是在`sort_buffer`和临时文件中执行的。但这个算法有一个问题，就是如果查询要返回的字段很多的话，那么`sort_buffer`里面要放的字段数太多，这样内存里能够同时放下的行数很少，要分成很多个临时文件，排序的性能会很差。

所以如果单行很大，这个方法效率不够好。

那么，如果MySQL认为排序的单行长度太大会怎么做呢？

接下来，我来修改一个参数，让MySQL采用另外一种算法。

```
SET max_length_for_sort_data = 16;
```

`max_length_for_sort_data`, 是MySQL中专门控制用于排序的行数据的长度的一个参数。它的意思是，如果单行的长度超过这个值，MySQL就认为单行太大，要换一个算法。

`city`、`name`、`age`这三个字段的定义总长度是36，我把`max_length_for_sort_data`设置为16，我们再来看看计算过程有什么改变。

新的算法放入`sort_buffer`的字段，只有要排序的列（即`name`字段）和主键`id`。

但这时，排序的结果就因为少了`city`和`age`字段的值，不能直接返回了，整个执行流程就变成如下所示的样子：

1. 初始化`sort_buffer`，确定放入两个字段，即`name`和`id`；
2. 从索引`city`找到第一个满足`city='杭州'`条件的主键`id`，也就是图中的`ID_X`；
3. 到主键`id`索引取出整行，取`name`、`id`这两个字段，存入`sort_buffer`中；
4. 从索引`city`取下一个记录的主键`id`；
5. 重复步骤3、4直到不满足`city='杭州'`条件为止，也就是图中的`ID_Y`；
6. 对`sort_buffer`中的数据按照字段`name`进行排序；
7. 遍历排序结果，取前1000行，并按照`id`的值回到原表中取出`city`、`name`和`age`三个字段返回给客户端。

这个执行流程的示意图如下，我把它称为`rowid`排序。

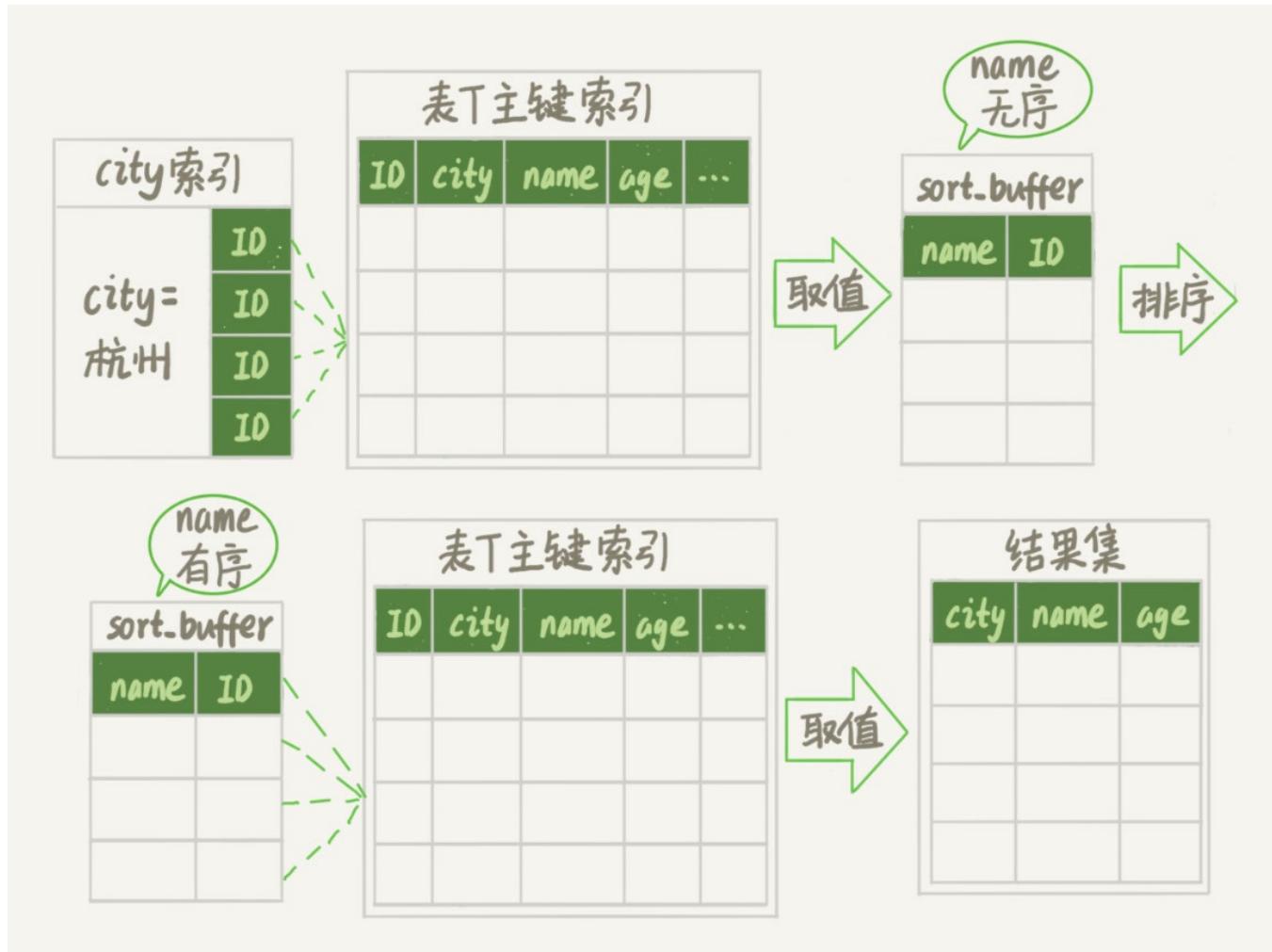


图5 rowid排序

对比图3的全字段排序流程图你会发现，**rowid**排序多访问了一次表t的主键索引，就是步骤7。

需要说明的是，最后的“结果集”是一个逻辑概念，实际上MySQL服务端从排序后的**sort_buffer**中依次取出**id**，然后到原表查到**city**、**name**和**age**这三个字段的结果，不需要在服务端再耗费内存存储结果，是直接返回给客户端的。

根据这个说明过程和图示，你可以想一下，这个时候执行**select @b-@a**，结果会是多少呢？

现在，我们就来看看结果有什么不同。

首先，图中的**examined_rows**的值还是4000，表示用于排序的数据是4000行。但是**select @b-@a**这个语句的值变成5000了。

因为这时候除了排序过程外，在排序完成后，还要根据**id**去原表取值。由于语句是**limit 1000**，因此会多读1000行。

```
"filesort_execution": [  
],  
"filesort_summary": {  
    "rows": 4000,  
    "examined_rows": 4000,  
    "number_of_tmp_files": 10,  
    "sort_buffer_size": 32728,  
    "sort_mode": "<sort_key, rowid>"  
}
```

图6 rowid排序的OPTIMIZER_TRACE部分输出

从OPTIMIZER_TRACE的结果中，你还能看到另外两个信息也变了。

- sort_mode变成了<sort_key, rowid>，表示参与排序的只有name和id这两个字段。
- number_of_tmp_files变成10了，是因为这时候参与排序的行数虽然仍然是4000行，但是每一行都变小了，因此需要排序的总数据量就变小了，需要的临时文件也相应地变少了。

全字段排序 VS rowid排序

我们来分析一下，从这两个执行流程里，还能得出什么结论。

如果MySQL实在是担心排序内存太小，会影响排序效率，才会采用rowid排序算法，这样排序过程中一次可以排序更多行，但是需要再回到原表去取数据。

如果MySQL认为内存足够大，会优先选择全字段排序，把需要的字段都放到sort_buffer中，这样排序后就会直接从内存里面返回查询结果了，不用再回到原表去取数据。

这也就体现了MySQL的一个设计思想：如果内存够，就要多利用内存，尽量减少磁盘访问。

对于InnoDB表来说，rowid排序会要求回表多造成磁盘读，因此不会被优先选择。

这个结论看上去有点废话的感觉，但是你要记住它，下一篇文章我们就会用到。

看到这里，你就了解了，MySQL做排序是一个成本比较高的操作。那么你会问，是不是所有的order by都需要排序操作呢？如果不排序就能得到正确的结果，那对系统的消耗会小很多，语句的执行时间也会变得更短。

其实，并不是所有的order by语句，都需要排序操作的。从上面分析的执行过程，我们可以看到，MySQL之所以需要生成临时表，并且在临时表上做排序操作，其原因是原来的数据都是无序的。

你可以设想下，如果能够保证从city这个索引上取出来的行，天然就是按照name递增排序的话，是不是就可以不用再排序了呢？

确实是这样的。

所以，我们可以在这个市民表上创建一个**city**和**name**的联合索引，对应的SQL语句是：

```
alter table t add index city_user(city, name);
```

作为与**city**索引的对比，我们来看看这个索引的示意图。

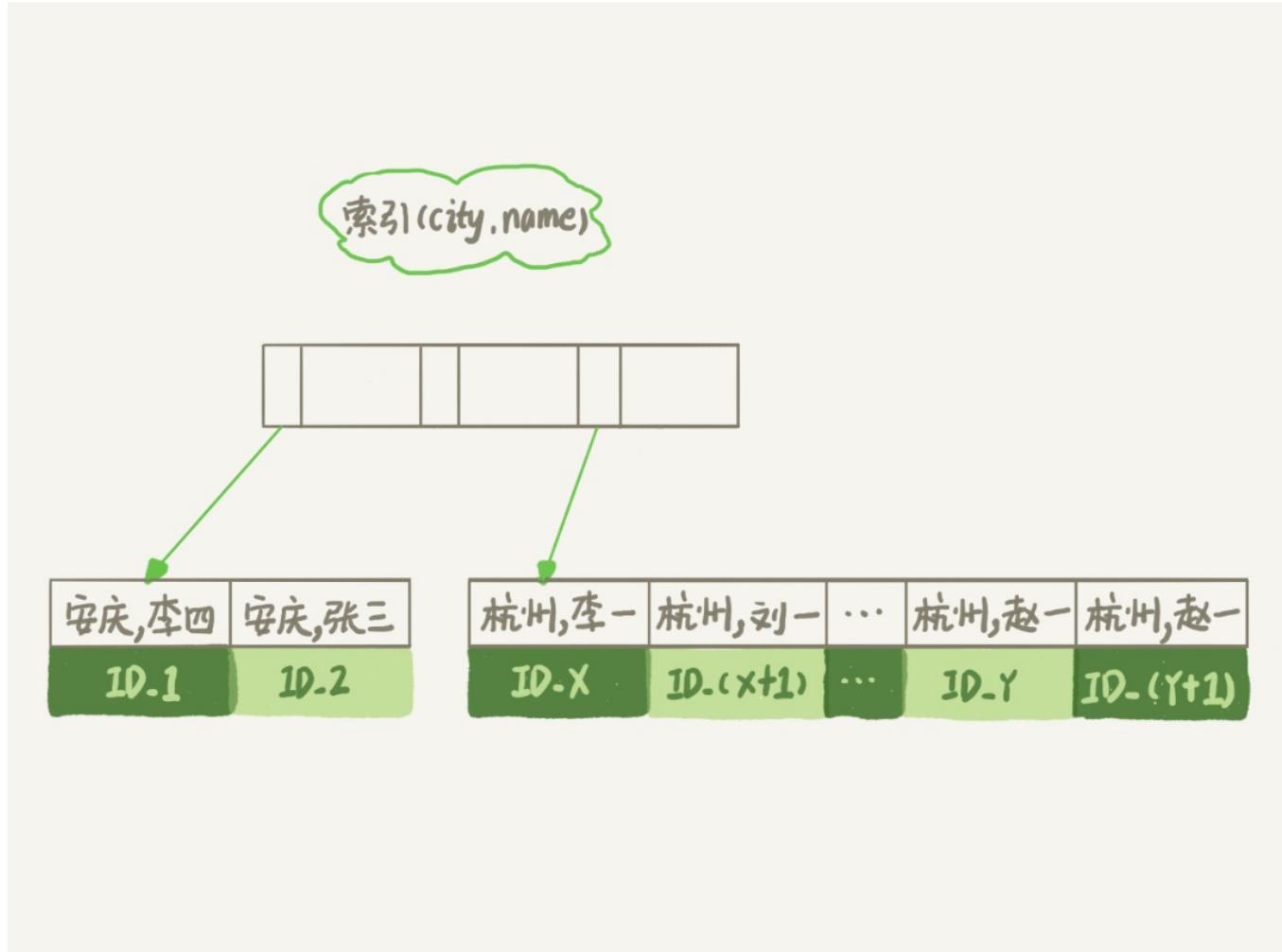


图7 city和name联合索引示意图

在这个索引里面，我们依然可以用树搜索的方式定位到第一个满足**city='杭州'**的记录，并且额外确保了，接下来按顺序取“下一条记录”的遍历过程中，只要**city**的值是杭州，**name**的值就一定是有序的。

这样整个查询过程的流程就变成了：

1. 从索引(**city, name**)找到第一个满足**city='杭州'**条件的主键**id**；
2. 到主键**id**索引取出整行，取**name**、**city**、**age**三个字段的值，作为结果集的一部分直接返回；
3. 从索引(**city, name**)取下一个记录主键**id**；

4. 重复步骤2、3，直到查到第1000条记录，或者是不满足city='杭州'条件时循环结束。

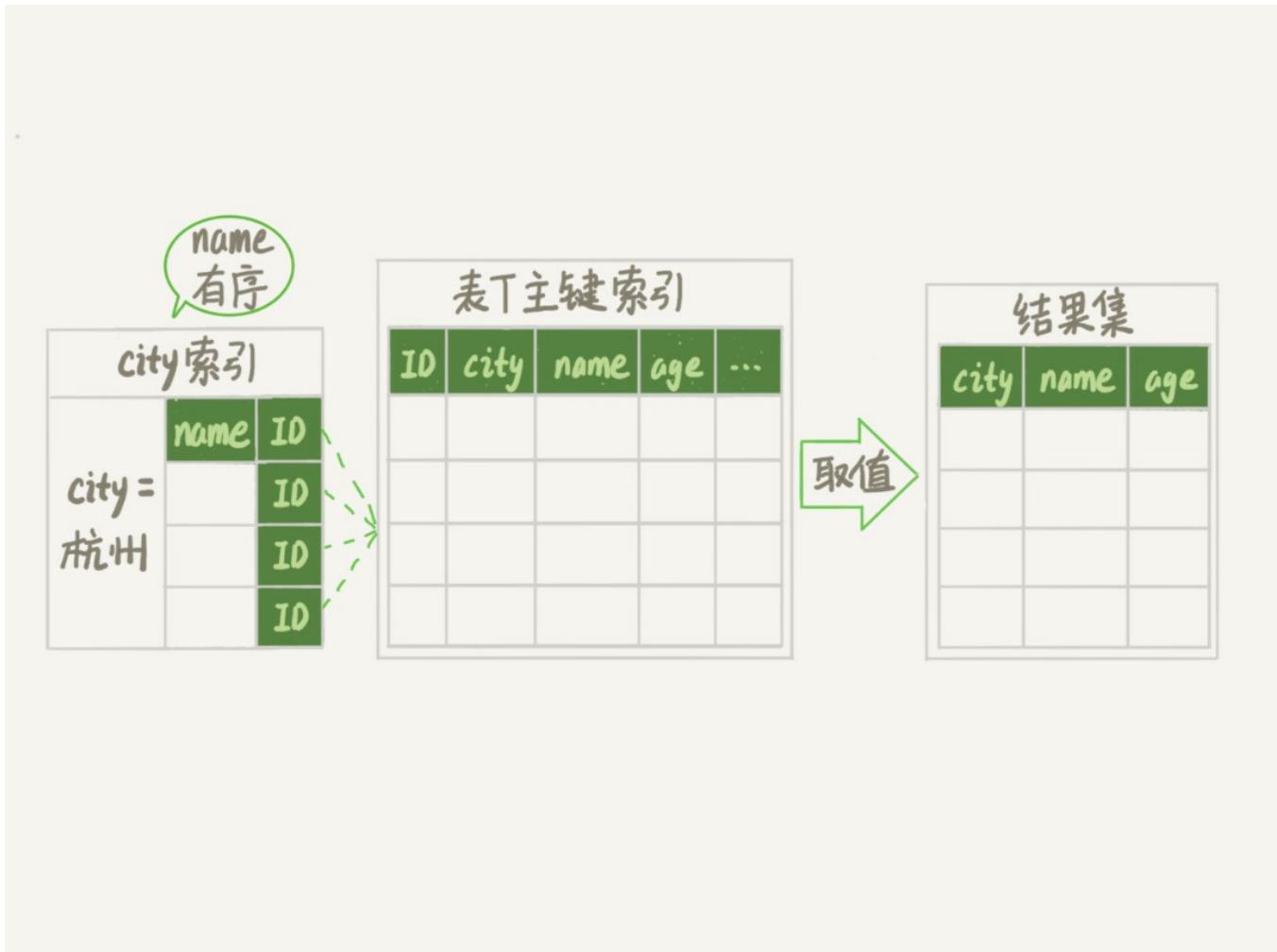


图8 引入(city,name)联合索引后，查询语句的执行计划

可以看到，这个查询过程不需要临时表，也不需要排序。接下来，我们用explain的结果来印证一下。

```
mysql> explain select city, name,age from T where city='杭州' order by name limit 1000;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | T | NULL | ref | city,city_user | city_user | 51 | const | 4000 | 100.00 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图9 引入(city,name)联合索引后，查询语句的执行计划

从图中可以看到，Extra字段中没有Using filesort了，也就是不需要排序了。而且由于(city,name)这个联合索引本身有序，所以这个查询也不用把4000行全都读一遍，只要找到满足条件的前1000条记录就可以退出了。也就是说，在我们这个例子里，只需要扫描1000次。

既然说到这里了，我们再往前讨论，这个语句的执行流程有没有可能进一步简化呢？不知道你还记不记得，我在第5篇文章[《深入浅出索引（下）》](#)中，和你介绍的覆盖索引。

这里我们可以再稍微复习一下。覆盖索引是指，索引上的信息足够满足查询请求，不需要再回到主键索引上去取数据。

按照覆盖索引的概念，我们可以再优化一下这个查询语句的执行流程。

针对这个查询，我们可以创建一个city、name和age的联合索引，对应的SQL语句就是：

```
alter table t add index city_user_age(city, name, age);
```

这时，对于city字段的值相同的行来说，还是按照name字段的值递增排序的，此时的查询语句也就不再需要排序了。这样整个查询语句的执行流程就变成了：

1. 从索引(city,name,age)找到第一个满足city='杭州'条件的记录，取出其中的city、name和age这三个字段的值，作为结果集的一部分直接返回；
2. 从索引(city,name,age)取下一个记录，同样取出这三个字段的值，作为结果集的一部分直接返回；
3. 重复执行步骤2，直到查到第1000条记录，或者是不满足city='杭州'条件时循环结束。



图10 引入(city,name,age)联合索引后，查询语句的执行流程

然后，我们再来看看explain的结果。

mysql> explain select city, name,age from T where city='杭州' order by name limit 1000;											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	T	NULL	ref	city,city_user,city_user_age	city_user_age	51	const	4000	100.00	Using where; Using index

图11 引入(city,name,age)联合索引后，查询语句的执行计划

可以看到，Extra字段里面多了“Using index”，表示的就是使用了覆盖索引，性能上会快很多。

当然，这里并不是说要为了每个查询能用上覆盖索引，就要把语句中涉及的字段都建上联合索引，毕竟索引还是有维护代价的。这是一个需要权衡的决定。

小结

今天这篇文章，我和你介绍了MySQL里面order by语句的几种算法流程。

在开发系统的时候，你总是不可避免地会使用到order by语句。你心里要清楚每个语句的排序逻辑是怎么实现的，还要能够分析出在最坏情况下，每个语句的执行对系统资源的消耗，这样才能做到下笔如有神，不犯低级错误。

最后，我给你留下一个思考题吧。

假设你的表里面已经有了city_name(city, name)这个联合索引，然后你要查杭州和苏州两个城市中所有的市民的姓名，并且按名字排序，显示前100条记录。如果SQL查询语句是这么写的：

```
mysql> select * from t where city in ('杭州','苏州') order by name limit 100;
```

那么，这个语句执行的时候会有排序过程吗，为什么？

如果业务端代码由你来开发，需要实现一个在数据库端不需要排序的方案，你会怎么实现呢？

进一步地，如果有分页需求，要显示第101页，也就是说语句最后要改成“limit 10000,100”，你的实现方法又会是什么呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，当MySQL去更新一行，但是要修改的值跟原来的值是相同的，这时候MySQL会真的去执行一次修改吗？还是看到值相同就直接返回呢？

这是第一次我们课后问题的三个选项都有同学选的，所以我要和你需要详细说明一下。

第一个选项是，MySQL读出数据，发现值与原来相同，不更新，直接返回，执行结束。这里我们可以用一个锁实验来确认。

假设，当前表t里的值是(1,2)。

session A	session B
begin; update t set a=2 where id=1;	
	update t set a=2 where id=1; (blocked)

图12 锁验证方式

session B的update语句被blocked了，加锁这个动作是InnoDB才能做的，所以排除选项1。

第二个选项是，MySQL调用了InnoDB引擎提供的接口，但是引擎发现值与原来相同，不更新，直接返回。有没有这种可能呢？这里我用一个可见性实验来确认。

假设当前表里的值是(1,2)。

session A	session B
begin; select * from t where id=1; /*返回(1,2)*/	
	update t set a=3 where id=1;
update t set a=3 where id=1; Query OK, 0 row affected (0.00 sec) Rows matched: 1 Changed: 0 Warnings: 0	
select * from t where id=1; /*返回(1,3)*/	

图13 可见性验证方式

session A的第二个select语句是一致性读（快照读），它是不能看见session B的更新的。

现在它返回的是(1,3)，表示它看见了某个新的版本，这个版本只能是session A自己的update语句做更新的时候生成。（如果你对这个逻辑有疑惑的话，可以回顾下第8篇文章[《事务到底是隔离的还是不隔离的？》](#)中的相关内容）

所以，我们上期思考题的答案应该是选项3，即：InnoDB认真执行了“把这个值修改成(1,2)”这个操作，该加锁的加锁，该更新的更新。

然后你会说，MySQL怎么这么笨，就不会更新前判断一下值是不是相同吗？如果判断一下，不就不用浪费InnoDB操作，多去更新一次了？

其实MySQL是确认了的。只是在这个语句里面，MySQL认为读出来的值，只有一个确定的(**id=1**)，而要写的是(**a=3**)，只从这两个信息是看不出来“不需要修改”的。

作为验证，你可以看一下下面这个例子。

session A	session B
begin; select * from t where id=1; /*返回 (1,2)*/	
	update t set a=3 where id=1;
update t set a=3 where id=1 and a=3;	
Query OK, 0 rows affected (0.00 sec) Rows matched: 1 Changed: 0 Warnings: 0	
select * from t where id=1; /*返回 (1,2)*/	

图14 可见性验证方式—对照

补充说明：

上面我们的验证结果都是在**binlog_format=statement**格式下进行的。

@didiren 补充了一个case，如果是**binlog_format=row** 并且**binlog_row_image=FULL**的时候，由于MySQL需要在**binlog**里面记录所有的字段，所以在读数据的时候就会把所有数据都读出来了。

根据上面说的规则，“既然读了数据，就会判断”，因此在这时候，**select * from t where id=1**，结果就是“返回 (1,2)”。

同理，如果是**binlog_row_image=NOBLOB**，会读出除**blob** 外的所有字段，在我们这个例子里，结果还是“返回 (1,2)”。

对应的代码如图15所示。这是MySQL 5.6版本引入的，在此之前我没有看过。所以，特此说明。

```
6570     switch (thd->variables.binlog_row_image)
6571     {
6572         case BINLOG_ROW_IMAGE_FULL:           //如果binlog是row格式，并且image=full
6573             if (s->primary_key < MAX_KEY)
6574                 bitmap_set_all(read_set);
6575             bitmap_set_all(write_set);
6576             break;                           //那么read_set设置为全1，表示所有的字段都要读
```

图15 binlog_row_image=FULL读字段逻辑

类似的，@mahonebags 同学提到了**timestamp**字段的问题。结论是：如果表中有**timestamp**字

段而且设置了自动更新的话，那么更新“别的字段”的时候，MySQL会读入所有涉及的字段，这样通过判断，就会发现不需要修改。

这两个点我会在后面讲更新性能的文章中再展开。

评论区留言点赞板：

@Gavin、@melon、@阿建 等同学提到了锁验证法；

@郭江伟 同学提到了两个点，都非常好，有去实际验证。结论是这样的：

第一，`hexdump`看出来没改应该是WAL机制生效了，要过一会儿，或者把库`shutdown`看看。

第二，`binlog`没写是MySQL Server层知道行的值没变，所以故意不写的，这个是在`row`格式下的策略。你可以把`binlog_format`改成`statement`再验证下。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Ding Qi, a man with glasses and a black shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45 讲' in large, bold, dark font, with the subtitle '从原理到实战，丁奇带你搞懂 MySQL' below it. On the far left is the '极客时间' logo. Below the title, the author's name '林晓斌' is listed, followed by '网名丁奇' and '前阿里资深技术专家'. At the bottom, there is a call-to-action: '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。'.

精选留言



某、人

20

回答下@发条橙子同学的问题：

问题一：

1)无条件查询如果只有`order by create_time`,即便`create_time`上有索引,也不会使用到。

因为优化器认为走二级索引再去回表成本比全表扫描排序更高。

所以选择走全表扫描,然后根据老师讲的两种方式选择一种来排序

2)无条件查询但是是`order by create_time limit m`.如果m值较小,是可以走索引的.

因为优化器认为根据索引有序性去回表查数据,然后得到m条数据,就可以终止循环,那么成本比

全表扫描小,则选择走二级索引。

即便没有二级索引,mysql针对order by limit也做了优化,采用堆排序。这部分老师明天会讲

问题二:

如果是group by a,a上不能使用索引的情况,是走rowid排序。

如果是group by limit,不能使用索引的情况,是走堆排序

如果是只有group by a,a上有索引的情况,又根据选取值不同,索引的扫描方式又有不同

`select * from t group by a`--走的是索引全扫描,至于这里为什么选择走索引全扫描,还需要老师解惑下

`select a from t group by a`--走的是索引松散扫描,也就说只需要扫描每组的第一行数据即可,不用扫描每一行的值

问题三:

`bigint`和`int`加数字都不影响能存储的值。

`bigint(1)`和`bigint(19)`都能存储 $2^{64}-1$ 范围内的值,`int`是 $2^{32}-1$ 。只是有些前端会根据括号里来截取显示而已。建议不加`varchar()`就必须带,因为`varchar()`括号里的数字代表能存多少字符。假设`varchar(2)`,就只能存两个字符,不管是中文还是英文。目前来看`varchar()`这个值可以设得稍稍大点,因为内存是按照实际的大小来分配内存空间的,不是按照值来预分配的。

老师我有几个问题:

1.我还是想在确认之前问的问题。一个长连接,一条sql申请了sort_buffer_size等一系列的会话级别的内存,sql成功执行完,该连接变为sleep状态。这些内存只是内容会被清空,但是占用的内存空间不会释放?

2.假设要给a值加1,执行器先找引擎取a=1的行,然后执行器给a+1,在调用接口写入a+1了数据。那么加锁不应该是执行器第一次去取数据时,引擎层就加该加的锁?为什么要等到第二次调用写入数据时,才加锁。第一次和第二次之间,难道不会被其他事务修改吗?如果没有锁保证

3.始终没太明白堆排序是采用的什么算法使得只需要对limit的数据进行排序就可以,而不是排序所有的数据在取前m条。--不过这里期待明天的文章

2018-12-20

| 作者回复

发条橙子同学的问题:

问题1:你回答得比我的答案还好! ↴

问题2:这个后面我们展开哈,要配图才能说得清楚 ↴

问题3:回答得也很好,需要注意的是255这个边界。小于255都需要一个字节记录长度,超过255就需要两个字节

你的问题: #好问题_#

1. 排序相关的内存排序后就free掉还给系统了

2. 读的时候加了写锁的

3. 堆排序要读所有行的,只读一次,我估计你已经理解对了 ↴

2018-12-20



didiren

6

刚才又测了一下，在`binlog-row-image=full`的情况下，第二次`update`是不写`redolog`的，说明`update`并没有发生

这样我就理解了，当`full`时，`mysql`需要读到在更新时读到`a`值，所以会判断`a`值不变，不需要更新，与你给出的`update t set a=3 where id=1 and a=3`原理相同，但`binlog-row-image`会影响查询结果还是会让人吃一惊

2018-12-19

| 作者回复

是的。

这个我也盲点了。

但是细想MySQL 选择这个策略又是合理的。

我需要再更新一下专栏内容

2018-12-19



null

2

re: 问题3:回答得也很好，需要注意的是255这个边界。小于255都需要一个字节记录长度，超过255就需要两个字节

11月过数据库设计方案，总监现场抛了一个问题，就是关于`varchar 255`的。现在回看，木有人回答到点上，都说是历史原因。

下回再问，就可以分享这一点了。呵呵(๑^◡^) 哇哈哈～

2018-12-21

| 作者回复

最怕的回答“历史原因”、“大家都这么做的所以...”、“别人要求的”

2018-12-21



老杨同志

11

1)

`mysql> select * from t where city in ('杭州','苏州') order by name limit 100;`

需要排序

原因是索引顺序城市、名称与单独按`name`排序的顺序不一致。

2) 如果不想`mysql`排序

方案a

可以执行两条语句

`select * from t where city = '杭州' limit 100;`

`select * from t where city = '苏州' limit 100;`

然后把200条记录在java中排序。

方案b

分别取前100，然后在数据端对200条数据进行排序。可以`sort buffer`就可以完成排序了。

少了一次应用程序与数据库的网络交互

```
select * from (
    select * from t where city = '杭州' limit 100
    union all
    select * from t where city = '苏州' limit 100
) as tt order by name limit 100
```

3) 对分页的优化。

没有特别好的办法。如果业务允许不提供排序功能，不提供查询最后一页，只能一页一页的翻，基本上前几页的数据已经满足客户需求。

为了意义不大的功能优化，可能会得不偿失。

如果一定要优化可以 `select id from t where city in ('杭州', '苏州') order by name limit 10000,100`

因为有city\name索引，上面的语句走覆盖索引就可以完成，不用回表。

最后使用 `select * from t where id in ()`; 取得结果

对于这个优化方法，我不好确定的是临界点，前几页直接查询就可以，最后几页使用这个优化方法。

但是中间的页码应该怎么选择不太清楚

2018-12-19

作者回复

从业务上砍掉功能，这个意识很好

2018-12-19



波波

笔记:

6

1. MySQL会为每个线程分配一个内存（`sort_buffer`）用于排序该内存大小为`sort_buffer_size`

1>如果排序的数据量小于`sort_buffer_size`, 排序将会在内存中完成

2>如果排序数据量很大，内存中无法存下这么多数据，则会使用磁盘临时文件来辅助排序，也称外部排序

3>在使用外部排序时，MySQL会分成好几份单独的临时文件用来存放排序后的数据，然后在将这些文件合并成一个大文件

2.mysql会通过遍历索引将满足条件的数据读取到`sort_buffer`, 并且按照排序字段进行快速排序

1>如果查询的字段不包含在辅助索引中，需要按照辅助索引记录的主键返回聚集索引取出所需字段

2>该方式会造成随机IO，在MySQL5.6提供了MRR的机制，会将辅助索引匹配记录的主键取出在内存中进行排序，然后在回表

3>按照情况建立联合索引来避免排序所带来的性能损耗，允许的情况下也可以建立覆盖索引来避免回表

全字段排序

- 1.通过索引将所需的字段全部读取到sort_buffer中
- 2.按照排序字段进行排序
- 3.将结果集返回给客户端

缺点：

- 1.造成sort_buffer中存放不下很多数据，因为除了排序字段还存放其他字段，对sort_buffer的利用效率不高
- 2.当所需排序数据量很大时，会有很多的临时文件，排序性能也会很差

优点：MySQL认为内存足够大时会优先选择全字段排序，因为这种方式比rowid排序避免了一次回表操作

rowid排序

- 1.通过控制排序的行数据的长度来让sort_buffer中尽可能多的存放数据，max_length_for_sort_data
- 2.只将需要排序的字段和主键读取到sort_buffer中，并按照排序字段进行排序
- 3.按照排序后的顺序，取id进行回表取出想要获取的数据
- 4.将结果集返回给客户端

优点：更好的利用内存的sort_buffer进行排序操作，尽量减少对磁盘的访问

缺点：回表的操作是随机IO，会造成大量的随机读，不一定就比全字段排序减少对磁盘的访问

- 3.按照排序的结果返回客户所取行数

2018-12-19

作者回复



2018-12-21



峰

3

由于city有两个值，相当于匹配到了索引树的两段区域，虽然各自都是按name排序，但整体需要做一次归并，当然只是limit100，所以够数就行。再然后如果需要不做排序，业务端就按city不同的取值查询两次，每次都limit100，然后业务端做归并处理喽。再然后要做分页的话，好吧，我的思路是先整出一张临时的结果表，`create table as select rownumber,* from t where city=x order by name`(写的不对哈，只是表达意思，rownumber为行数，并为主键)然后直接从这张表中按rownumber进行分页查询就好。

2018-12-19

作者回复

分页这个再考虑考虑哈

2018-12-19



赵海亮

2

老师你好，全字段排序那一节，我做了实验，我的排序缓存大小是1M，examined rows 是7715892，查询的三个字段都有数据，那么如果这些数据都放到缓存应该需要 $(4+8+11) * 7715892$ 等于160M，但是我看了都没有用到临时表，这是为什么？

```
CREATE TABLE `phone_call_logs` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键ID',
  `city_id` int(11) NOT NULL DEFAULT '11',
  `call_sender` varchar(40) DEFAULT NULL COMMENT '电话主叫号码',
  `phone_id` bigint(20) NOT NULL DEFAULT '0' COMMENT '手机id',
  PRIMARY KEY (`id`),
  KEY `idx_city` (`city_id`)
) ENGINE=InnoDB AUTO_INCREMENT=64551193;
-----sort_buffer_size=1M-----
root:(none)> show variables like 'sort_buffer_size';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| sort_buffer_size | 1048576 |
+-----+-----+
1 row in set (0.00 sec)
-----查询sql-----
select city_id,phone_id,call_sender from phone_call_logs where city_id=11 order by phone_id desc limit 1000;
```

-----执行计划结果-----

```
"filesort_priority_queue_optimization": {
  "limit": 1000,
  "rows_estimate": 146364461,
  "row_size": 146,
  "memory_available": 1048576,
  "chosen": true
},
"filesort_execution": [
],
"filesort_summary": {
  "rows": 1001,
  "examined_rows": 7715892,
  "number_of_tmp_files": 0,
  "sort_buffer_size": 154160,
  "sort_mode": "<sort_key, additional_fields>"}
```

2018-12-19

作者回复

好问题，明天见！

(明天的一篇也是跟排序有关的哦)

2018-12-20



didiren

2

感谢！针对我之前提出的疑问，我又详细的做了实验，发现一个新的问题，我感觉是个bug，希望解答

SessionA

```
mysql> show variables like '%binlog_row_image%';
+ Variable_name | Value |
| binlog_row_image | FULL |
mysql> create table t (id int not null primary key auto_increment,
-> a int default null)
-> engine=innodb;
mysql> insert into t values(1,2);
mysql> set tx_isolation = 'repeatable-read';
mysql> begin;
mysql> select * from t where id = 1;
+ id | a |
| 1 | 2 |
```

此时在另一个SessionB执行update t set a=3 where id = 1;成功更新一条记录。通过show engine innodb status看，Log sequence number 2573458

然后在SessionA继续。。

```
mysql> update t set a=3 where id = 1;
Rows matched: 1 Changed: 0 Warnings: 0
Log sequence number 2573467
mysql> select * from t where id = 1;
+ id | a |
| 1 | 2 |
```

这里与你给出的答案里的实验结果不同

可以看到redolog是记录了第二次的update的，但是select却没有看到更新后的值，于是我又换了一个平时测试用的实例，同样的步骤却得到了与你的答案相同的结果

然后我对比了2个实例的参数，发现当binlog-row-image=minimal时第二次查询结果a=3，当binlog-row-image=full时第二次查询结果a=2，而且不论哪个参数，redolog都会因为SessionA的update增长，说明redolog都做了记录，update是发生了的，但是binlog-row-image参数会影响查询结果，难以理解，我用的mysql版本是官方的5.7.13

下面是binlog-row-image = minimal的实验结果

```
mysql> set binlog_row_image=MINIMAL;
mysql> drop table t;
```

```
mysql> create table t (id int not null primary key auto_increment,
-> a int default null)
-> engine=innodb;
insert into t values(1,2);
mysql> insert into t values(1,2);
mysql> set tx_isolation = 'repeatable-read';
mysql> begin;
mysql> select * from t where id = 1;
| id | a |
| 1  | 2  |
```

此时在另一个SessionB执行update t set a=3 where id = 1;成功更新一条记录。

```
mysql> update t set a=3 where id = 1;
Rows matched: 1 Changed: 0 Warnings: 0
mysql> select * from t where id = 1;
| id | a |
| 1  | 3  |
```

2018-12-19

| 作者回复

！ ! !

你说的对

我验证的是**statement**格式。

MySQL 看来选了不错吧路径。

这个我之前真不知道||

多谢

2018-12-19



cyberbit

2

1.不会有排序，这种情况属于《高性能mysql》里提到的“in技法”，符合索引的最左原则，是2个等值查询，可以用到右边的索引列。

2.分页查询，可以用延迟关联来优化：

```
select * from t join
(select id from t where city in('杭州','苏州') order by name limit 10000,100) t_id
on t.id=t_id.id;
```

2018-12-19



尘封

2

请问，第7步中遍历排序结果，取前 1000 行，并按照 id 的值回到原表中取出 city、name 和 age 三个字段返回给客户端：这里会把id再进行排序吗？转随机io为顺序io？

2018-12-19

| 作者回复

要是排序就结果不符合order by 的语义逻辑了...

2018-12-19



进击的菜鸟

1

关于上期问题里的最后一个例子不太明白，还请老师指点一下。按说在更新操作的时候应该是当前读，那么应该能读到id=1 and a = 3的记录并修改。那么为什么再select还会查到a = 2。难道是即便update但是where条件也是快照读？但是如果这样那么幻读的问题不就不会存在了吗？（B insert了一条记录，此时A范围update后再select会把B insert的语句查出来）

2019-02-02

| 作者回复

你是说图14这里对吧，

这里update语句自己是当前读，但是它没有更新数据；

所以之后的查询还是看不到(1,3)这个版本。

好问题！

2019-02-02



发条橙子。

1

老师，接前面 create_time的回答。语句确实是 select * from t order by create_time desc ;

老师是指 优化器会根据 order by create_time 来选择使用 create_time 索引么

我之前误以为优化器是根据 where 后面的字段条件来选择索引，所以上面那条语句没有where的时候我就想当然地以为不会走索引。看来是自己跳进了一个大坑里面！

另：我之前在本地建了张表加了20w数据，用explain 查了一次，发现走的是全表没有走索引，老师说会走索引。我想了一下，可能是统计的数据有误的缘故，用 analyze table重新统计，再次查询果然走了索引。！

2018-12-20

| 作者回复

嗯 where和 order都会共同影响哦，今天这篇你要再看看最后加了联合索引以后，语句的执行逻辑

Analyze table 立功啦！

2018-12-20



发条橙子。

1

正好有个 `order by` 使用场景，有个页面，需要按数据插入时间倒序来查看一张记录表的信息，因为除了分页的参数，没有其他 `where` 的条件，所以除了主键外没有其他索引。

这时候 DBA 让我给 `create_time` 创建索引，说是按照顺序排列，查询会增快。这篇文章看完后，让我感觉实际上创建 `create_time` 索引是没用的。

因为查询本身并没有用到 `create_time` 索引，实际上查询的步骤是：

1. 初始化 `sort_buffer` 内存
2. 因为没索引，所以扫出全表的数据到 `sort_buffer` 中
2. 如果内存够则直接内存按时间排序
3. 如果内存不够则按数据量分成不同文件分别按时间排序后整合
4. 根据数量分页查询数量回聚集索引中用 ID 查询数据
5. 返回

所以我分析 `create_time` 索引应该不需要创建。反而增加了维护成本

问题一：这种无条件查列表页除了全表扫还有其他建立索引的办法么

问题二：如果加入 `group by`，数据该如何走

问题三：老师之后的文章会有讲解 `bigint(20)`、`tinyint(2)`、`varchar(32)` 这种后面带数字与不带数字有何区别的文章么。每次建字段都会考虑长度，但实际却不知道他有何作用

2018-12-20

| 作者回复

你说的这样场景，加上 `create_time` 索引的话，是可以加速的呀，

语句是这样吗？`select * from t order by create_time desk limit 100?` 如果是这样，创建索引有用的。

问题二后面会有文章会说哈

问题三 嗯，这个也会安排文章说到

2018-12-20



明亮

1

需要排序，可以将原来的索引中**name**字段放前面，**city**字段放后面，来建索引就可以了

2018-12-19

| 作者回复

这样不太好哈，变成全索引扫描了

2018-12-19



进击的菜鸟

0

图14那个疑问明白了，是因为**where**条件中存在**update**的值InnoDB认为值一致所以没有修改，从而导致A的一致性视图中看不到B的修改。

这篇又看了一遍，还有个疑问，想请老师解答一下。

1.asc和desc会影响使用索引排序吗？

2.如果采用**rowid**也无法放入排序字段还是会转用磁盘排序吧。

2019-02-06

| 作者回复

新年快乐

1. 不影响

2. 再小也用**rowid**，对，会转成磁盘排序

2019-02-06



frogman

0

你好，不知道可否讲一下**Group By**怎样运用索引来优化呢？文档说只有**MIN**和**MAX**的情况下索引才能提高效率，但是我认为**COUNT**应该也行吧。如 **select name count(*) from table where name=\$name group by address;** 如果建立**id**和**name**的联合索引，效率会提高吗。

2019-02-03

| 作者回复

你的理解对的。

我们后面有一篇会讲到**group by**

2019-02-03



Crayon

0

老师，您好，我想问一下对于评论区中的无条件排序问题，知道了走索引和不走索引的两种情况。

在我的测试中，无条件多字段排序，即使加了**LIMIT**，也会走全表扫描，这是为什么呢。排序的字段们都加了普通索引

2019-01-30

| 作者回复

试下，是不是其实你不加**limit**反而会用索引

2019-01-30



过去、今日

0

老师，您好！大概是这样子的，`app`表的数据量是10w，`api_form`的数据量大概50w以后还是增加，`app`表的索引为`app_key,api_form`索引为`app_key, modified`。但是从执行计划上看`Extra`的值为`Using where Using fileSort`

```
select * from app t1,api_form t2 where t1.app_key=t2.app_key and t2.status=1 order by t2.modified
```

2019-01-27

| 作者回复

`api_form`索引为`app_key, modified`

是联合索引吗，如果是可以`force`一下这个索引

你还是两个表（脱敏后）发一下表结构，还有这个语句`explain`的结果

2019-01-28



过去、今日

0

老师有个问题

```
select * from a t1,b t2 where t1.id = t2.id order by t1.created
```

即时`a`表和`b`表都创建了索引，但是通过执行计划看都是`using filesort`

2019-01-27

| 作者回复

贴一下你这两个表的表结构，和`explain`的结果。

这个语句要没有`filesort`的话，需要`t1`当驱动表，并且`t1`有`(id, created)`联合索引

2019-01-27



Wesley

0

表T有`id name city age`三列，对`name, city`两列做联合索引。

执行`explain select name, city from T where name like '%' and city ='BJ' order by name limit 10 ;` `rows`的值永远都和`limit`的值一样。虽然`rows`是预估的值，但是也不会每次和要`limit`的值一样吧

2019-01-23

| 作者回复

给一下你的复现步骤哈，不太可能“永远一样的”

2019-01-23

16 | “order by”是怎么工作的？

2018-12-19 林晓斌



在你开发应用的时候，一定会经常碰到需要根据指定的字段排序来显示结果的需求。还是以我们前面举例用过的市民表为例，假设你要查询城市是“杭州”的所有名字，并且按照姓名排序返回前1000个人的姓名、年龄。

假设这个表的部分定义是这样的：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `city` varchar(16) NOT NULL,
  `name` varchar(16) NOT NULL,
  `age` int(11) NOT NULL,
  `addr` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `city` (`city`)
) ENGINE=InnoDB;
```

这时，你的SQL语句可以这么写：

```
select city,name,age from t where city='杭州' order by name limit 1000 ;
```

这个语句看上去逻辑很清晰，但是你了解它的执行流程吗？今天，我就和你聊聊这个语句是怎么执行的，以及有什么参数会影响执行的行为。

全字段排序

前面我们介绍过索引，所以你现在就很清楚了，为避免全表扫描，我们需要在city字段加上索引。

在city字段上创建索引之后，我们用explain命令来看看这个语句的执行情况。

```
mysql> explain select city, name, age from T where city='杭州' order by name limit 1000;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | partitions | type | possible_keys | key | key_len | ref  | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE     | T      | NULL       | ref  | city          | city | 51    | const | 4000 | 100.00  | Using index condition; Using filesort
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图1 使用explain命令查看语句的执行情况

Extra这个字段中的“Using filesort”表示的就是需要排序，MySQL会给每个线程分配一块内存用于排序，称为sort_buffer。

为了说明这个SQL查询语句的执行过程，我们先来看一下city这个索引的示意图。

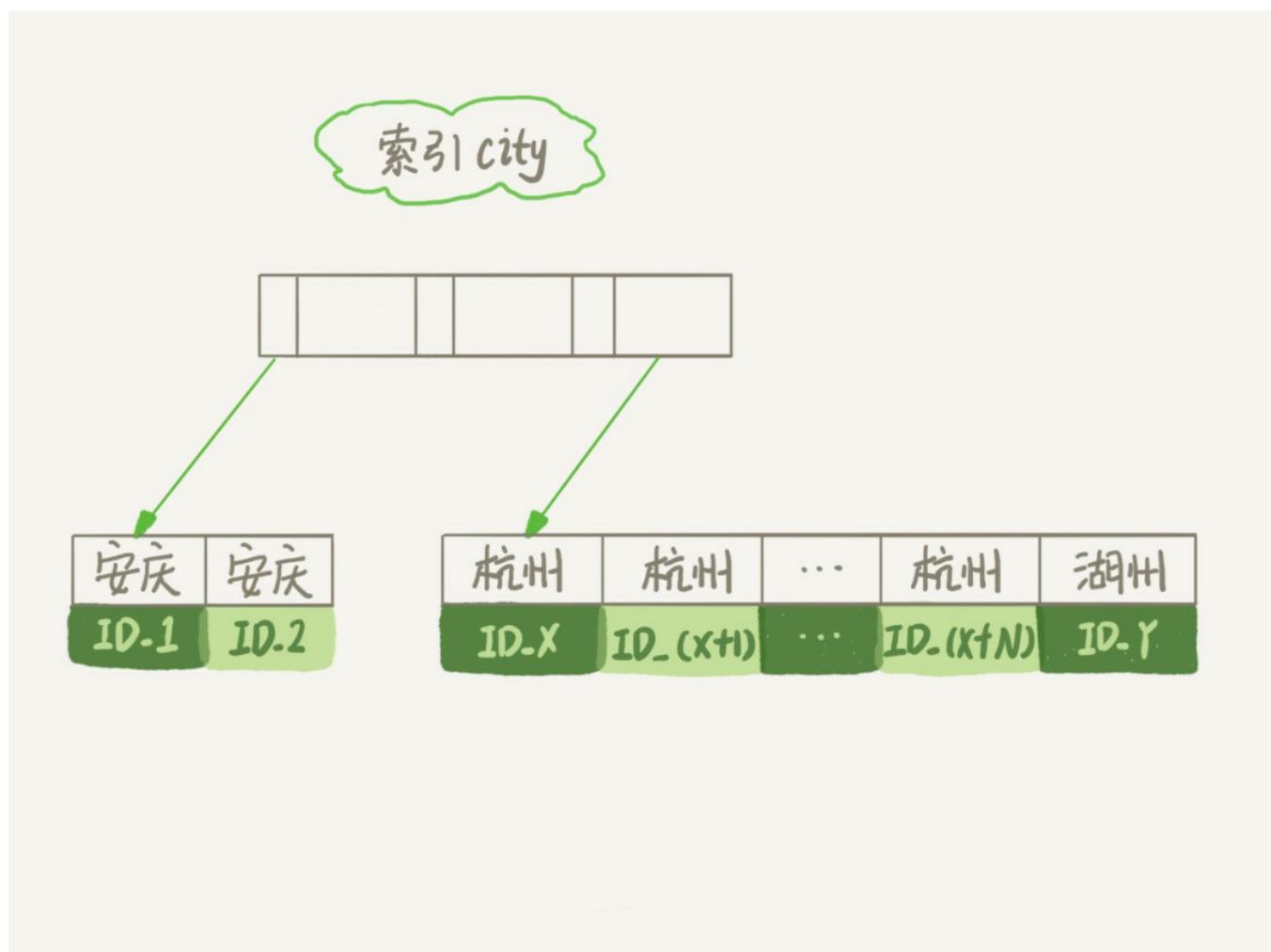


图2 city字段的索引示意图

从图中可以看到，满足city='杭州'条件的行，是从ID_X到ID_(X+N)的这些记录。

通常情况下，这个语句执行流程如下所示：

1. 初始化sort_buffer，确定放入name、city、age这三个字段；
2. 从索引city找到第一个满足city='杭州'条件的主键id，也就是图中的ID_X；
3. 到主键id索引取出整行，取name、city、age三个字段的值，存入sort_buffer中；
4. 从索引city取下一个记录的主键id；
5. 重复步骤3、4直到city的值不满足查询条件为止，对应的主键id也就是图中的ID_Y；
6. 对sort_buffer中的数据按照字段name做快速排序；
7. 按照排序结果取前1000行返回给客户端。

我们暂且把这个排序过程，称为全字段排序，执行流程的示意图如下所示，下一篇文章中我们还会用到这个排序。

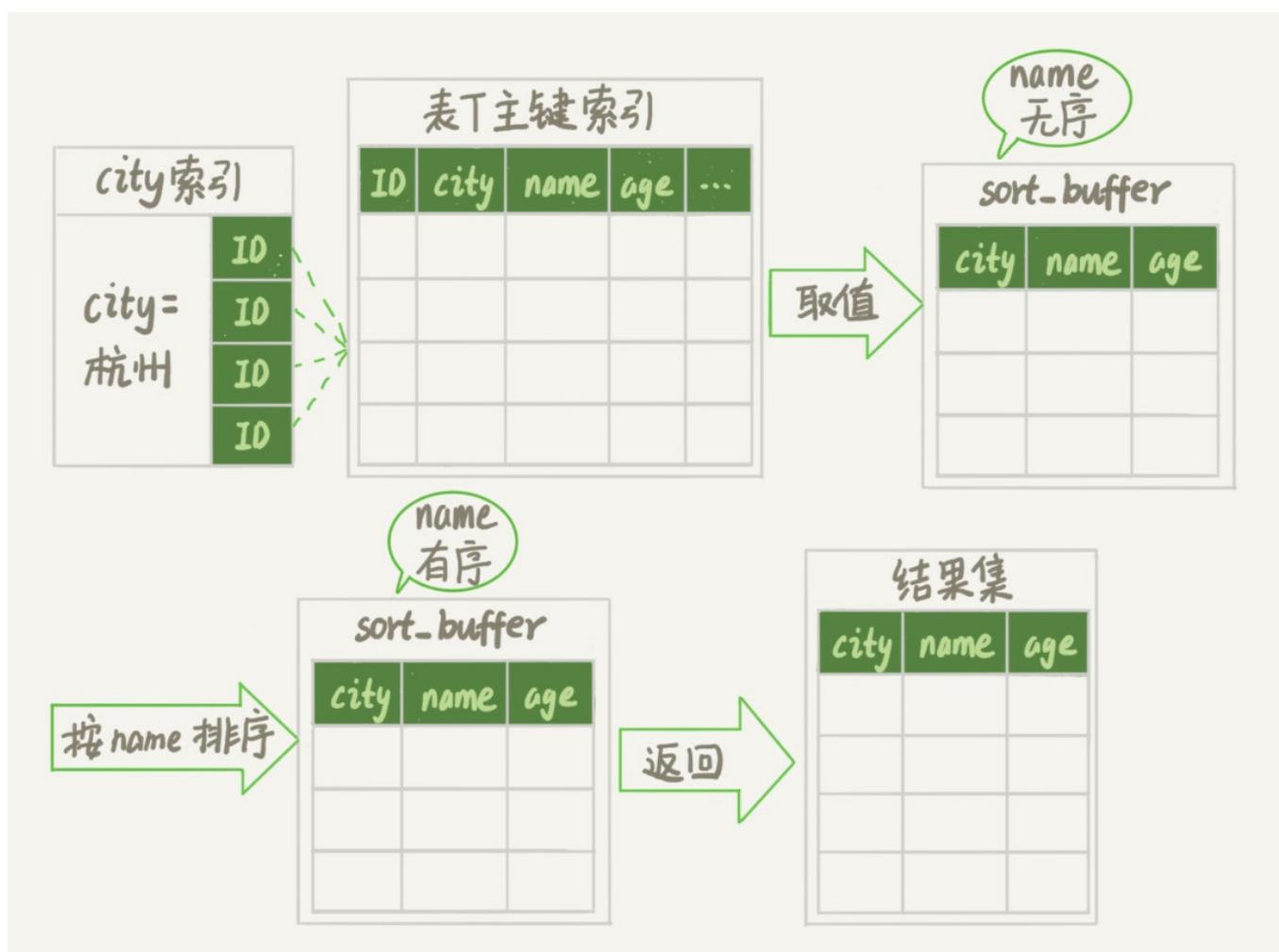


图3 全字段排序

图中“按name排序”这个动作，可能在内存中完成，也可能需要使用外部排序，这取决于排序所需的内存和参数sort_buffer_size。

sort_buffer_size，就是MySQL为排序开辟的内存（sort_buffer）的大小。如果要排序的数据量小于sort_buffer_size，排序就在内存中完成。但如果排序数据量太大，内存放不下，则不得不利用磁盘临时文件辅助排序。

你可以用下面介绍的方法，来确定一个排序语句是否使用了临时文件。

```
/* 打开optimizer_trace，只对本线程有效 */
SET optimizer_trace='enabled=on';

/* @a保存Innodb_rows_read的初始值 */
select VARIABLE_VALUE into @a from performance_schema.session_status where variable_name = 'Innodb_rows_read';

/* 执行语句 */
select city, name,age from t where city='杭州' order by name limit 1000;

/* 查看 OPTIMIZER_TRACE 输出 */
SELECT * FROM `information_schema`.`OPTIMIZER_TRACE` \G

/* @b保存Innodb_rows_read的当前值 */
select VARIABLE_VALUE into @b from performance_schema.session_status where variable_name = 'Innodb_rows_read';

/* 计算Innodb_rows_read差值 */
select @b-@a;
```

这个方法是通过查看 OPTIMIZER_TRACE 的结果来确认的，你可以从 number_of_tmp_files 中看到是否使用了临时文件。

```
"filesort_execution": [
],
"filesort_summary": {
    "rows": 4000,
    "examined_rows": 4000,
    "number_of_tmp_files": 12,
    "sort_buffer_size": 32684,
    "sort_mode": "<sort_key, packed_additional_fields>"
}
```

图4 全排序的OPTIMIZER_TRACE部分结果

`number_of_tmp_files`表示的是，排序过程中使用的临时文件数。你一定奇怪，为什么需要12个文件？内存放不下时，就需要使用外部排序，外部排序一般使用归并排序算法。可以这么简单理解，MySQL将需要排序的数据分成12份，每一份单独排序后存在这些临时文件中。然后把这12个有序文件再合并成一个有序的大文件。

如果`sort_buffer_size`超过了需要排序的数据量的大小，`number_of_tmp_files`就是0，表示排序可以直接在内存中完成。

否则就需要放在临时文件中排序。`sort_buffer_size`越小，需要分成的份数越多，`number_of_tmp_files`的值就越大。

接下来，我再和你解释一下图4中其他两个值的意思。

我们的示例表中有4000条满足`city='杭州'`的记录，所以你可以看到`examined_rows=4000`，表示参与排序的行数是4000行。

`sort_mode`里面的`packed_additional_fields`的意思是，排序过程对字符串做了“紧凑”处理。即使`name`字段的定义是`varchar(16)`，在排序过程中还是要按照实际长度来分配空间的。

同时，最后一个查询语句`select @b-@a`的返回结果是4000，表示整个执行过程只扫描了4000行。

这里需要注意的是，为了避免对结论造成干扰，我把`internal_tmp_disk_storage_engine`设置成MyISAM。否则，`select @b-@a`的结果会显示为4001。

这是因为查询OPTIMIZER_TRACE这个表时，需要用到临时表，而`internal_tmp_disk_storage_engine`的默认值是InnoDB。如果使用的是InnoDB引擎的话，把数据从临时表取出来的时候，会让`Innodb_rows_read`的值加1。

rowid排序

在上面这个算法过程里面，只对原表的数据读了一遍，剩下的操作都是在`sort_buffer`和临时文件中执行的。但这个算法有一个问题，就是如果查询要返回的字段很多的话，那么`sort_buffer`里面要放的字段数太多，这样内存里能够同时放下的行数很少，要分成很多个临时文件，排序的性能会很差。

所以如果单行很大，这个方法效率不够好。

那么，如果MySQL认为排序的单行长度太大会怎么做呢？

接下来，我来修改一个参数，让MySQL采用另外一种算法。

```
SET max_length_for_sort_data = 16;
```

`max_length_for_sort_data`, 是MySQL中专门控制用于排序的行数据的长度的一个参数。它的意思是，如果单行的长度超过这个值，MySQL就认为单行太大，要换一个算法。

`city`、`name`、`age`这三个字段的定义总长度是36，我把`max_length_for_sort_data`设置为16，我们再来看看计算过程有什么改变。

新的算法放入`sort_buffer`的字段，只有要排序的列（即`name`字段）和主键`id`。

但这时，排序的结果就因为少了`city`和`age`字段的值，不能直接返回了，整个执行流程就变成如下所示的样子：

1. 初始化`sort_buffer`，确定放入两个字段，即`name`和`id`；
2. 从索引`city`找到第一个满足`city='杭州'`条件的主键`id`，也就是图中的`ID_X`；
3. 到主键`id`索引取出整行，取`name`、`id`这两个字段，存入`sort_buffer`中；
4. 从索引`city`取下一个记录的主键`id`；
5. 重复步骤3、4直到不满足`city='杭州'`条件为止，也就是图中的`ID_Y`；
6. 对`sort_buffer`中的数据按照字段`name`进行排序；
7. 遍历排序结果，取前1000行，并按照`id`的值回到原表中取出`city`、`name`和`age`三个字段返回给客户端。

这个执行流程的示意图如下，我把它称为`rowid`排序。

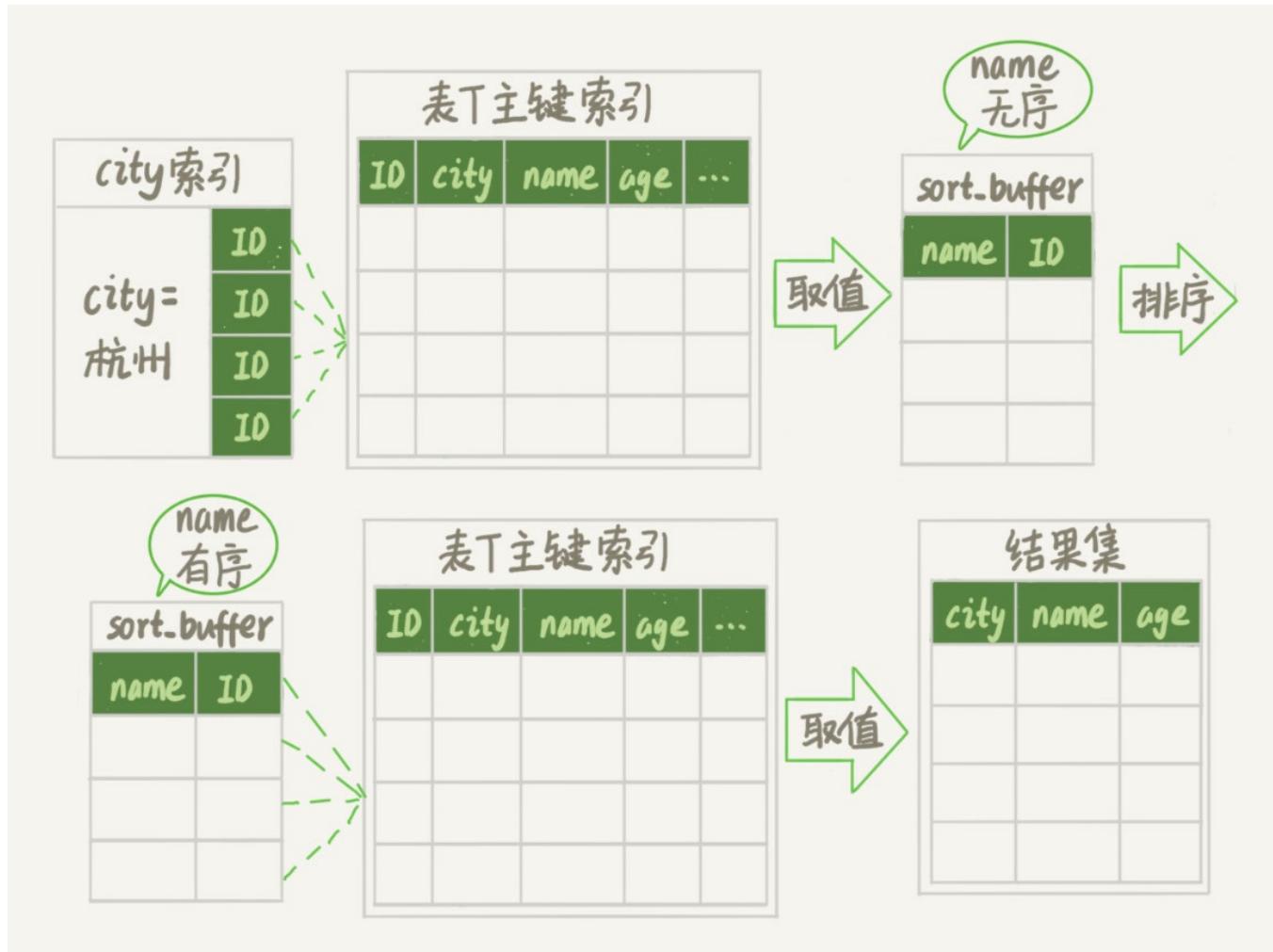


图5 rowid排序

对比图3的全字段排序流程图你会发现，**rowid**排序多访问了一次表t的主键索引，就是步骤7。

需要说明的是，最后的“结果集”是一个逻辑概念，实际上MySQL服务端从排序后的**sort_buffer**中依次取出**id**，然后到原表查到**city**、**name**和**age**这三个字段的结果，不需要在服务端再耗费内存存储结果，是直接返回给客户端的。

根据这个说明过程和图示，你可以想一下，这个时候执行**select @b-@a**，结果会是多少呢？

现在，我们就来看看结果有什么不同。

首先，图中的**examined_rows**的值还是4000，表示用于排序的数据是4000行。但是**select @b-@a**这个语句的值变成5000了。

因为这时候除了排序过程外，在排序完成后，还要根据**id**去原表取值。由于语句是**limit 1000**，因此会多读1000行。

```
"filesort_execution": [  
],  
"filesort_summary": {  
    "rows": 4000,  
    "examined_rows": 4000,  
    "number_of_tmp_files": 10,  
    "sort_buffer_size": 32728,  
    "sort_mode": "<sort_key, rowid>"  
}
```

图6 rowid排序的OPTIMIZER_TRACE部分输出

从OPTIMIZER_TRACE的结果中，你还能看到另外两个信息也变了。

- sort_mode变成了<sort_key, rowid>，表示参与排序的只有name和id这两个字段。
- number_of_tmp_files变成10了，是因为这时候参与排序的行数虽然仍然是4000行，但是每一行都变小了，因此需要排序的总数据量就变小了，需要的临时文件也相应地变少了。

全字段排序 VS rowid排序

我们来分析一下，从这两个执行流程里，还能得出什么结论。

如果MySQL实在是担心排序内存太小，会影响排序效率，才会采用rowid排序算法，这样排序过程中一次可以排序更多行，但是需要再回到原表去取数据。

如果MySQL认为内存足够大，会优先选择全字段排序，把需要的字段都放到sort_buffer中，这样排序后就会直接从内存里面返回查询结果了，不用再回到原表去取数据。

这也就体现了MySQL的一个设计思想：如果内存够，就要多利用内存，尽量减少磁盘访问。

对于InnoDB表来说，rowid排序会要求回表多造成磁盘读，因此不会被优先选择。

这个结论看上去有点废话的感觉，但是你要记住它，下一篇文章我们就会用到。

看到这里，你就了解了，MySQL做排序是一个成本比较高的操作。那么你会问，是不是所有的order by都需要排序操作呢？如果不排序就能得到正确的结果，那对系统的消耗会小很多，语句的执行时间也会变得更短。

其实，并不是所有的order by语句，都需要排序操作的。从上面分析的执行过程，我们可以看到，MySQL之所以需要生成临时表，并且在临时表上做排序操作，其原因是原来的数据都是无序的。

你可以设想下，如果能够保证从city这个索引上取出来的行，天然就是按照name递增排序的话，是不是就可以不用再排序了呢？

确实是这样的。

所以，我们可以在这个市民表上创建一个**city**和**name**的联合索引，对应的SQL语句是：

```
alter table t add index city_user(city, name);
```

作为与**city**索引的对比，我们来看看这个索引的示意图。

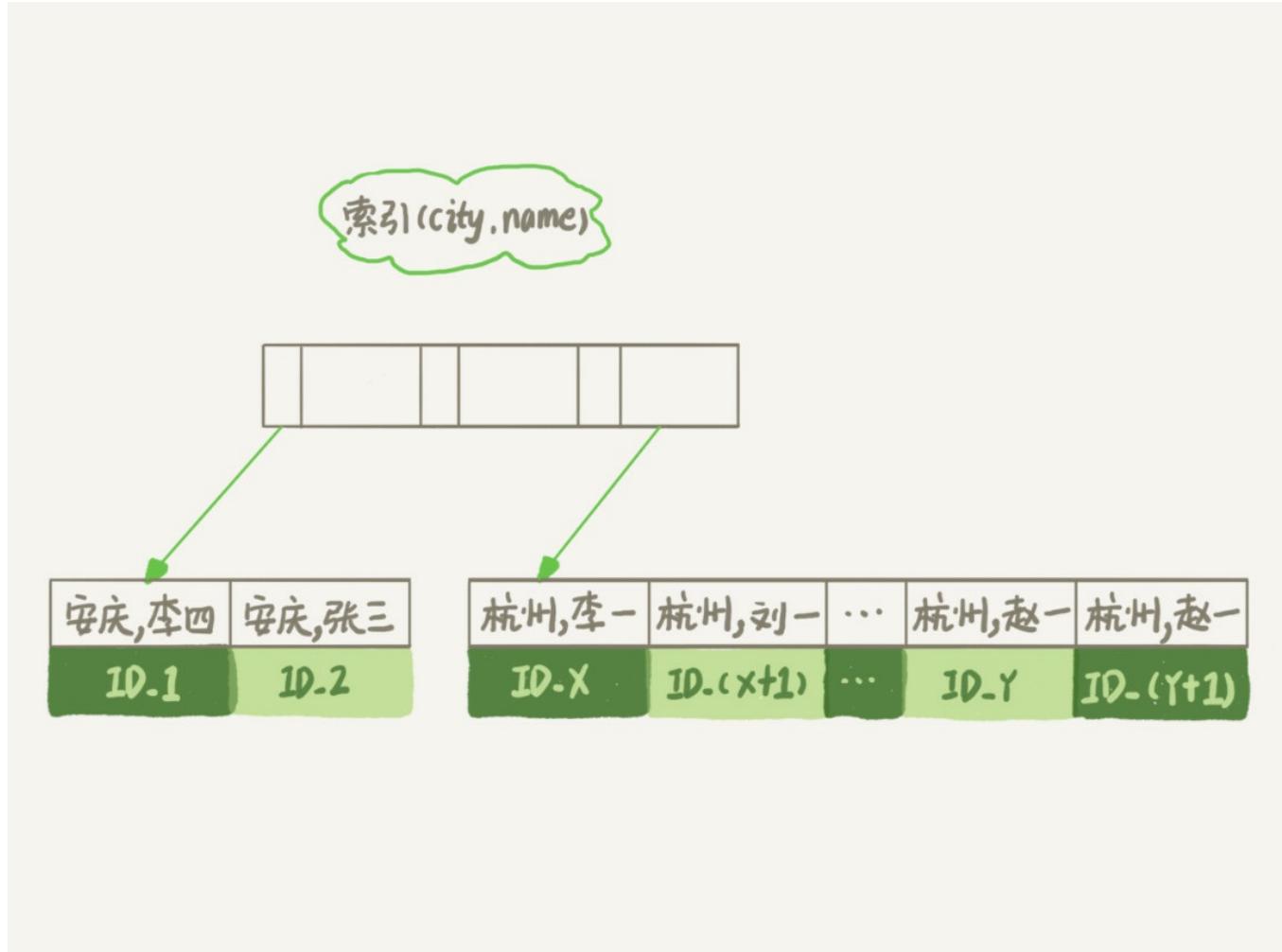


图7 city和name联合索引示意图

在这个索引里面，我们依然可以用树搜索的方式定位到第一个满足**city='杭州'**的记录，并且额外确保了，接下来按顺序取“下一条记录”的遍历过程中，只要**city**的值是杭州，**name**的值就一定是有序的。

这样整个查询过程的流程就变成了：

1. 从索引(**city,name**)找到第一个满足**city='杭州'**条件的主键**id**;
2. 到主键**id**索引取出整行，取**name**、**city**、**age**三个字段的值，作为结果集的一部分直接返回；
3. 从索引(**city,name**)取下一个记录主键**id**;

4. 重复步骤2、3，直到查到第1000条记录，或者是不满足city='杭州'条件时循环结束。

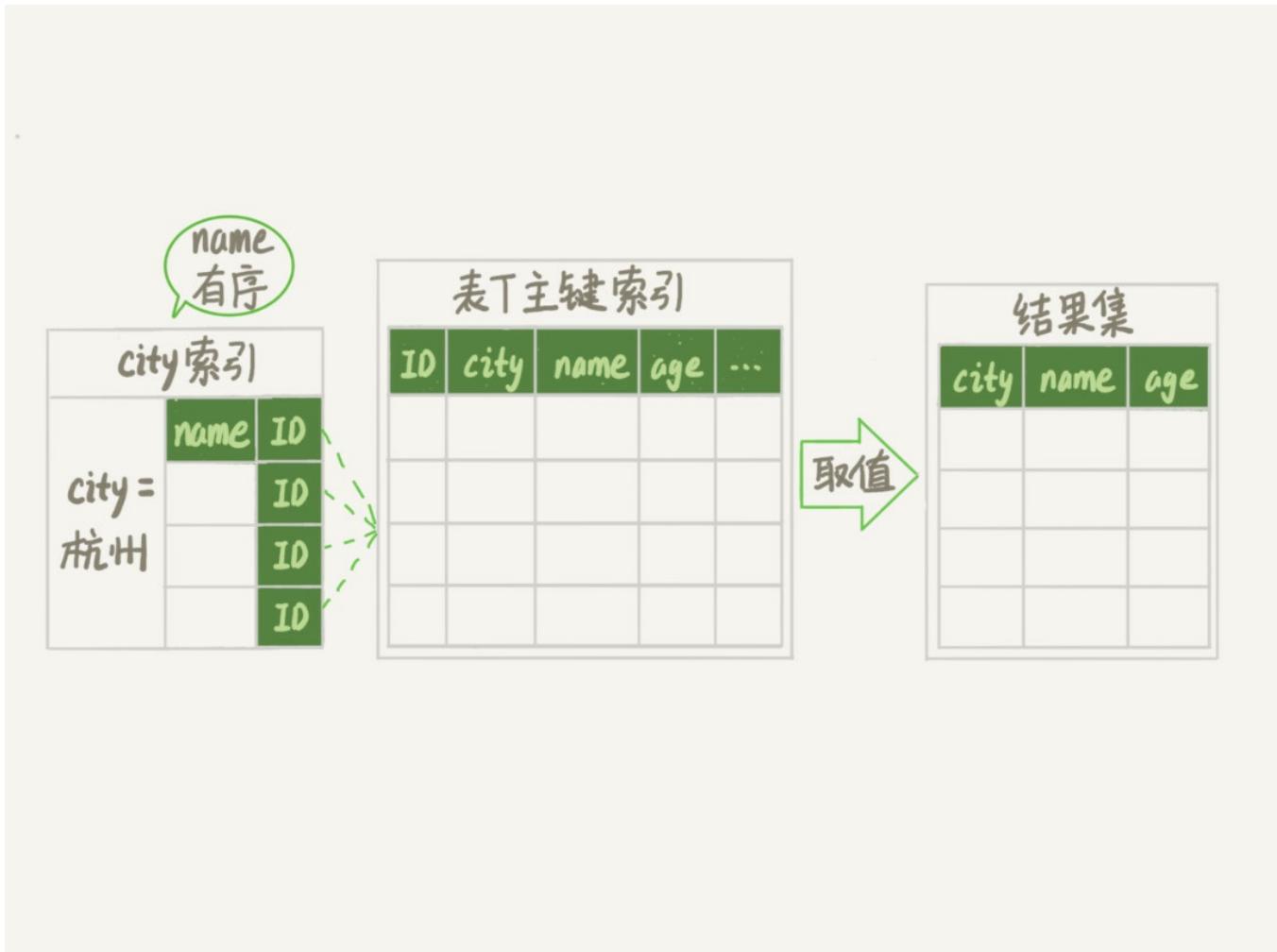


图8 引入(city,name)联合索引后，查询语句的执行计划

可以看到，这个查询过程不需要临时表，也不需要排序。接下来，我们用explain的结果来印证一下。

```
mysql> explain select city, name,age from T where city='杭州' order by name limit 1000;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | T | NULL | ref | city,city_user | city_user | 51 | const | 4000 | 100.00 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图9 引入(city,name)联合索引后，查询语句的执行计划

从图中可以看到，Extra字段中没有Using filesort了，也就是不需要排序了。而且由于(city,name)这个联合索引本身有序，所以这个查询也不用把4000行全都读一遍，只要找到满足条件的前1000条记录就可以退出了。也就是说，在我们这个例子里，只需要扫描1000次。

既然说到这里了，我们再往前讨论，这个语句的执行流程有没有可能进一步简化呢？不知道你还记不记得，我在第5篇文章[《深入浅出索引（下）》](#)中，和你介绍的覆盖索引。

这里我们可以再稍微复习一下。覆盖索引是指，索引上的信息足够满足查询请求，不需要再回到主键索引上去取数据。

按照覆盖索引的概念，我们可以再优化一下这个查询语句的执行流程。

针对这个查询，我们可以创建一个city、name和age的联合索引，对应的SQL语句就是：

```
alter table t add index city_user_age(city, name, age);
```

这时，对于city字段的值相同的行来说，还是按照name字段的值递增排序的，此时的查询语句也就不再需要排序了。这样整个查询语句的执行流程就变成了：

1. 从索引(city,name,age)找到第一个满足city='杭州'条件的记录，取出其中的city、name和age这三个字段的值，作为结果集的一部分直接返回；
2. 从索引(city,name,age)取下一个记录，同样取出这三个字段的值，作为结果集的一部分直接返回；
3. 重复执行步骤2，直到查到第1000条记录，或者是不满足city='杭州'条件时循环结束。



图10 引入(city,name,age)联合索引后，查询语句的执行流程

然后，我们再来看看explain的结果。

mysql> explain select city, name,age from T where city='杭州' order by name limit 1000;											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	T	NULL	ref	city,city_user,city_user_age	city_user_age	51	const	4000	100.00	Using where; Using index

图11 引入(city,name,age)联合索引后，查询语句的执行计划

可以看到，Extra字段里面多了“Using index”，表示的就是使用了覆盖索引，性能上会快很多。

当然，这里并不是说要为了每个查询能用上覆盖索引，就要把语句中涉及的字段都建上联合索引，毕竟索引还是有维护代价的。这是一个需要权衡的决定。

小结

今天这篇文章，我和你介绍了MySQL里面order by语句的几种算法流程。

在开发系统的时候，你总是不可避免地会使用到order by语句。你心里要清楚每个语句的排序逻辑是怎么实现的，还要能够分析出在最坏情况下，每个语句的执行对系统资源的消耗，这样才能做到下笔如有神，不犯低级错误。

最后，我给你留下一个思考题吧。

假设你的表里面已经有了city_name(city, name)这个联合索引，然后你要查杭州和苏州两个城市中所有的市民的姓名，并且按名字排序，显示前100条记录。如果SQL查询语句是这么写的：

```
mysql> select * from t where city in ('杭州','苏州') order by name limit 100;
```

那么，这个语句执行的时候会有排序过程吗，为什么？

如果业务端代码由你来开发，需要实现一个在数据库端不需要排序的方案，你会怎么实现呢？

进一步地，如果有分页需求，要显示第101页，也就是说语句最后要改成“limit 10000,100”，你的实现方法又会是什么呢？

你可以把你的思考和观点写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，当MySQL去更新一行，但是要修改的值跟原来的值是相同的，这时候MySQL会真的去执行一次修改吗？还是看到值相同就直接返回呢？

这是第一次我们课后问题的三个选项都有同学选的，所以我要和你需要详细说明一下。

第一个选项是，MySQL读出数据，发现值与原来相同，不更新，直接返回，执行结束。这里我们可以用一个锁实验来确认。

假设，当前表t里的值是(1,2)。

session A	session B
begin; update t set a=2 where id=1;	
	update t set a=2 where id=1; (blocked)

图12 锁验证方式

session B的update语句被blocked了，加锁这个动作是InnoDB才能做的，所以排除选项1。

第二个选项是，MySQL调用了InnoDB引擎提供的接口，但是引擎发现值与原来相同，不更新，直接返回。有没有这种可能呢？这里我用一个可见性实验来确认。

假设当前表里的值是(1,2)。

session A	session B
begin; select * from t where id=1; /*返回(1,2)*/	
	update t set a=3 where id=1;
update t set a=3 where id=1;	
Query OK, 0 row affected (0.00 sec) Rows matched: 1 Changed: 0 Warnings: 0	
select * from t where id=1; /*返回(1,3)*/	

图13 可见性验证方式

session A的第二个select语句是一致性读（快照读），它是不能看见session B的更新的。

现在它返回的是(1,3)，表示它看见了某个新的版本，这个版本只能是session A自己的update语句做更新的时候生成。（如果你对这个逻辑有疑惑的话，可以回顾下第8篇文章[《事务到底是隔离的还是不隔离的？》](#)中的相关内容）

所以，我们上期思考题的答案应该是选项3，即：InnoDB认真执行了“把这个值修改成(1,2)”这个操作，该加锁的加锁，该更新的更新。

然后你会说，MySQL怎么这么笨，就不会更新前判断一下值是不是相同吗？如果判断一下，不就不用浪费InnoDB操作，多去更新一次了？

其实MySQL是确认了的。只是在这个语句里面，MySQL认为读出来的值，只有一个确定的(**id=1**)，而要写的是(**a=3**)，只从这两个信息是看不出来“不需要修改”的。

作为验证，你可以看一下下面这个例子。

session A	session B
begin; select * from t where id=1; /*返回 (1,2)*/	
	update t set a=3 where id=1;
update t set a=3 where id=1 and a=3;	
Query OK, 0 rows affected (0.00 sec) Rows matched: 1 Changed: 0 Warnings: 0	
select * from t where id=1; /*返回 (1,2)*/	

图14 可见性验证方式—对照

补充说明：

上面我们的验证结果都是在**binlog_format=statement**格式下进行的。

@didiren 补充了一个case，如果是**binlog_format=row** 并且**binlog_row_image=FULL**的时候，由于MySQL需要在**binlog**里面记录所有的字段，所以在读数据的时候就会把所有数据都读出来了。

根据上面说的规则，“既然读了数据，就会判断”，因此在这时候，**select * from t where id=1**，结果就是“返回 (1,2)”。

同理，如果是**binlog_row_image=NOBLOB**，会读出除**blob** 外的所有字段，在我们这个例子里，结果还是“返回 (1,2)”。

对应的代码如图15所示。这是MySQL 5.6版本引入的，在此之前我没有看过。所以，特此说明。

```
6570     switch (thd->variables.binlog_row_image)
6571     {
6572         case BINLOG_ROW_IMAGE_FULL:           如果binlog是row格式，并且image=full
6573             if (s->primary_key < MAX_KEY)
6574                 bitmap_set_all(read_set);
6575             bitmap_set_all(write_set);
6576             break;                           那么 read_set设置为全1，表示所有的字段都要读
```

图15 binlog_row_image=FULL读字段逻辑

类似的，@mahonebags 同学提到了**timestamp**字段的问题。结论是：如果表中有**timestamp**字

段而且设置了自动更新的话，那么更新“别的字段”的时候，MySQL会读入所有涉及的字段，这样通过判断，就会发现不需要修改。

这两个点我会在后面讲更新性能的文章中再展开。

评论区留言点赞板：

@Gavin、@melon、@阿建 等同学提到了锁验证法；

@郭江伟 同学提到了两个点，都非常好，有去实际验证。结论是这样的：

第一，`hexdump`看出来没改应该是WAL机制生效了，要过一会儿，或者把库`shutdown`看看。

第二，`binlog`没写是MySQL Server层知道行的值没变，所以故意不写的，这个是在row格式下的策略。你可以把`binlog_format`改成`statement`再验证下。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Ding Qi, a man with short dark hair and glasses, wearing a black button-down shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45 讲' in large, bold, dark font, with the subtitle '从原理到实战，丁奇带你搞懂 MySQL' below it. To the left of the title is the '极客时间' logo, which consists of a stylized orange 'Q' icon followed by the text '极客时间'. Below the title, there is a section for the author: '林晓斌' (Ling Xiaobin) with the note '网名丁奇 前阿里资深技术专家'. At the bottom of the image, there is a call-to-action: '新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。'.

17 | 如何正确地显示随机消息？

2018-12-21 林晓斌



我在上一篇文章，为你讲解完**order by**语句的几种执行模式后，就想到了之前一个做英语学习App的朋友碰到过的一个性能问题。今天这篇文章，我就从这个性能问题说起，和你说说MySQL中的另外一种排序需求，希望能够加深你对MySQL排序逻辑的理解。

这个英语学习App首页有一个随机显示单词的功能，也就是根据每个用户的级别有一个单词表，然后这个用户每次访问首页的时候，都会随机滚动显示三个单词。他们发现随着单词表变大，选单词这个逻辑变得越来越慢，甚至影响到了首页的打开速度。

现在，如果让你来设计这个SQL语句，你会怎么写呢？

为了便于理解，我对这个例子进行了简化：去掉每个级别的用户都有一个对应的单词表这个逻辑，直接就是从一个单词表中随机选出三个单词。这个表的建表语句和初始数据的命令如下：

```

mysql> CREATE TABLE `words` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `word` varchar(64) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;

delimiter ;;

create procedure idata()
begin
    declare i int;
    set i=0;
    while i<10000 do
        insert into words(word) values(concat(char(97+(i div 1000)), char(97+(i % 1000 div 100)), char(97+(i % 100 div 10)), char(97+i % 10)));
        set i=i+1;
    end while;
end;;
delimiter ;

call idata();

```

为了便于量化说明，我在这个表里面插入了10000行记录。接下来，我们就一起看看要随机选择3个单词，有什么方法实现，存在什么问题以及如何改进。

内存临时表

首先，你会想到用`order by rand()`来实现这个逻辑。

```
mysql> select word from words order by rand() limit 3;
```

这个语句的意思很直白，随机排序取前3个。虽然这个SQL语句写法很简单，但执行流程却有点复杂的。

我们先用`explain`命令来看看这个语句的执行情况。

```
mysql> explain select word from words order by rand() limit 3;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | words | NULL       | ALL   | NULL          | NULL | NULL    | NULL | 9980 |    100.00 | Using temporary; Using filesort |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图1 使用explain命令查看语句的执行情况

Extra字段显示Using temporary, 表示的是需要使用临时表; Using filesort, 表示的是需要执行排序操作。

因此这个Extra的意思就是，需要临时表，并且需要在临时表上排序。

这里，你可以先回顾一下[上一篇文章](#)中全字段排序和rowid排序的内容。我把上一篇文章的两个流程图贴过来，方便你复习。

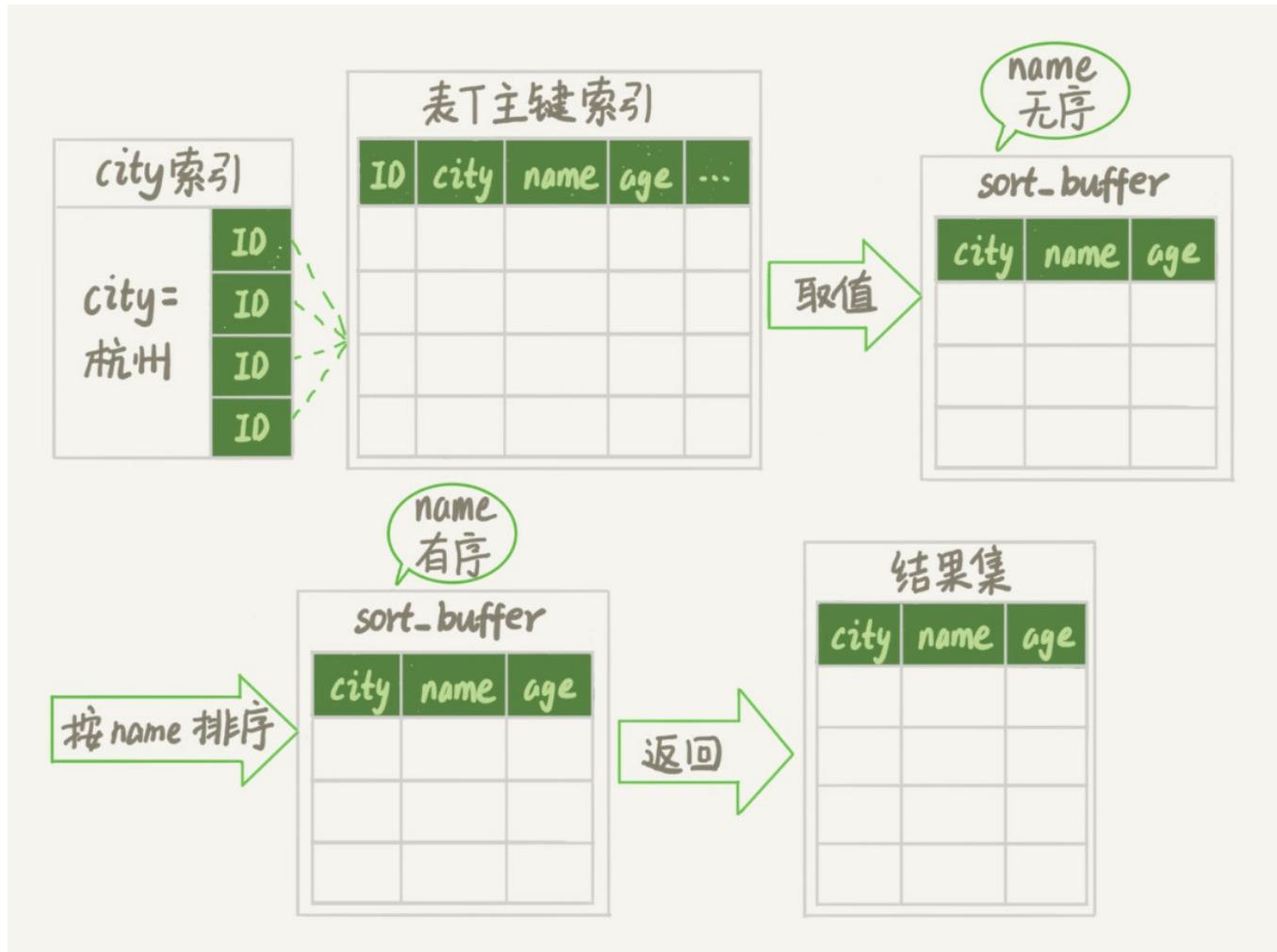


图2 全字段排序

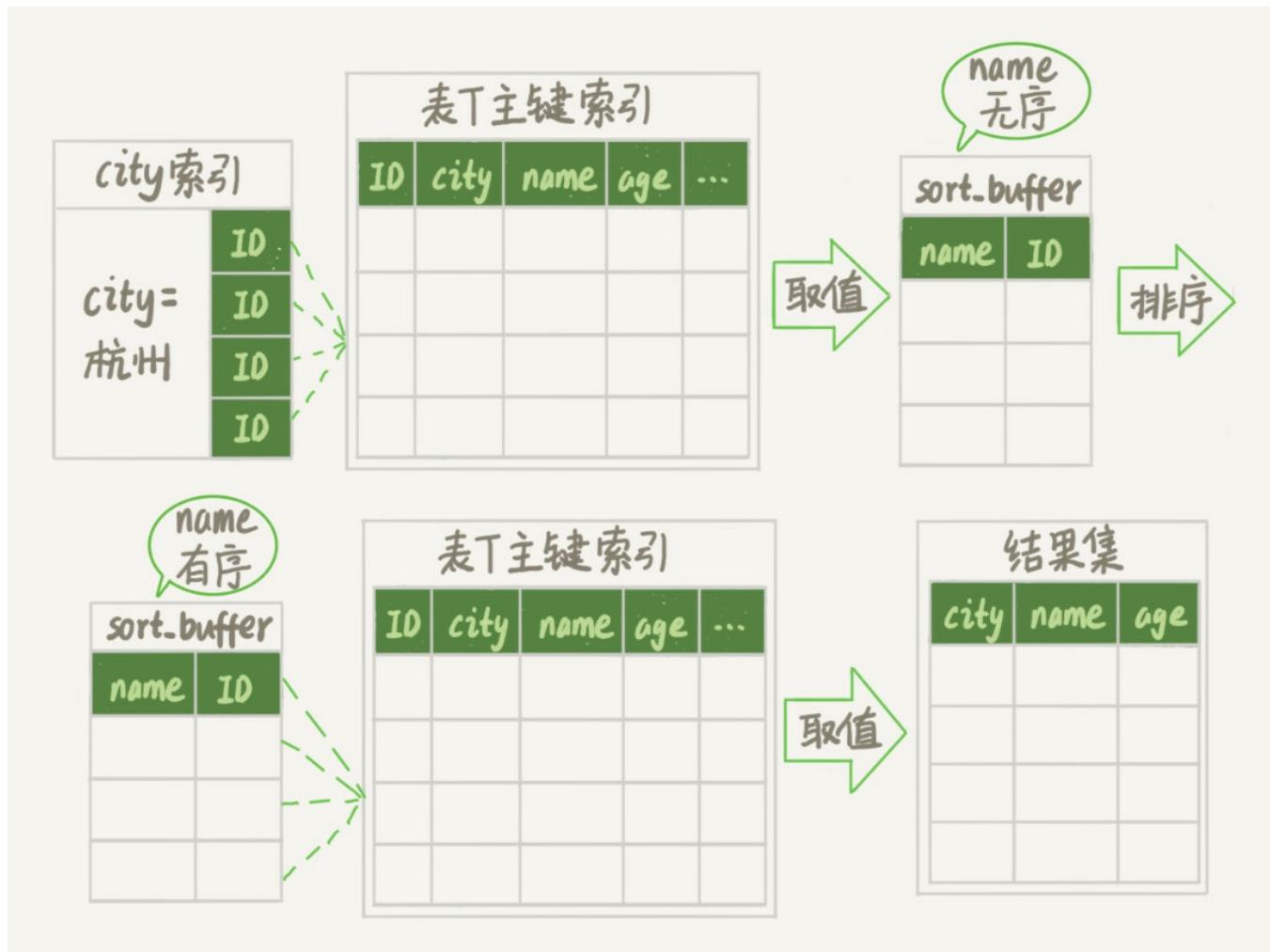


图3 rowid排序

然后，我再问你一个问题，你觉得对于临时内存表的排序来说，它会选择哪一种算法呢？回顾一下上一篇文章的一个结论：对于**InnoDB表**来说，执行全字段排序会减少磁盘访问，因此会被优先选择。

我强调了“**InnoDB表**”，你肯定想到了，对于内存表，回表过程只是简单地根据数据行的位置，直接访问内存得到数据，根本不会导致多访问磁盘。优化器没有了这一层顾虑，那么它会优先考虑的，就是用于排序的行越少越好了，所以，MySQL这时就会选择**rowid**排序。

理解了这个算法选择的逻辑，我们再来看看语句的执行流程。同时，通过今天的这个例子，我们来尝试分析一下语句的扫描行数。

这条语句的执行流程是这样的：

1. 创建一个临时表。这个临时表使用的是**memory**引擎，表里有两个字段，第一个字段是**double**类型，为了后面描述方便，记为字段R，第二个字段是**varchar(64)**类型，记为字段W。并且，这个表没有建索引。
2. 从**words**表中，按主键顺序取出所有的**word**值。对于每一个**word**值，调用**rand()**函数生成一个大于0小于1的随机小数，并把这个随机小数和**word**分别存入临时表的R和W字段中，到

此，扫描行数是**10000**。

3. 现在临时表有**10000**行数据了，接下来你要在这个没有索引的内存临时表上，按照字段**R**排序。
4. 初始化 **sort_buffer**。**sort_buffer**中有两个字段，一个是**double**类型，另一个是整型。
5. 从内存临时表中一行一行地取出**R**值和位置信息（我后面会和你解释这里为什么是“位置信息”），分别存入**sort_buffer**中的两个字段里。这个过程要对内存临时表做全表扫描，此时扫描行数增加**10000**，变成了**20000**。
6. 在**sort_buffer**中根据**R**的值进行排序。注意，这个过程没有涉及到表操作，所以不会增加扫描行数。
7. 排序完成后，取出前三个结果的位置信息，依次到内存临时表中取出**word**值，返回给客户端。这个过程中，访问了表的三行数据，总扫描行数变成了**20003**。

接下来，我们通过慢查询日志（**slow log**）来验证一下我们分析得到的扫描行数是否正确。

```
# Query_time: 0.900376 Lock_time: 0.000347 Rows_sent: 3 Rows_examined: 20003
SET timestamp=1541402277;
select word from words order by rand() limit 3;
```

其中，**Rows_examined: 20003**就表示这个语句执行过程中扫描了**20003**行，也就验证了我们分析得出的结论。

这里插一句题外话，在平时学习概念的过程中，你可以经常这样做，先通过原理分析算出扫描行数，然后再通过查看慢查询日志，来验证自己的结论。我自己就是经常这么做，这个过程很有趣，分析对了开心，分析错了但是弄清楚了也很开心。

现在，我来把完整的排序执行流程图画出来。

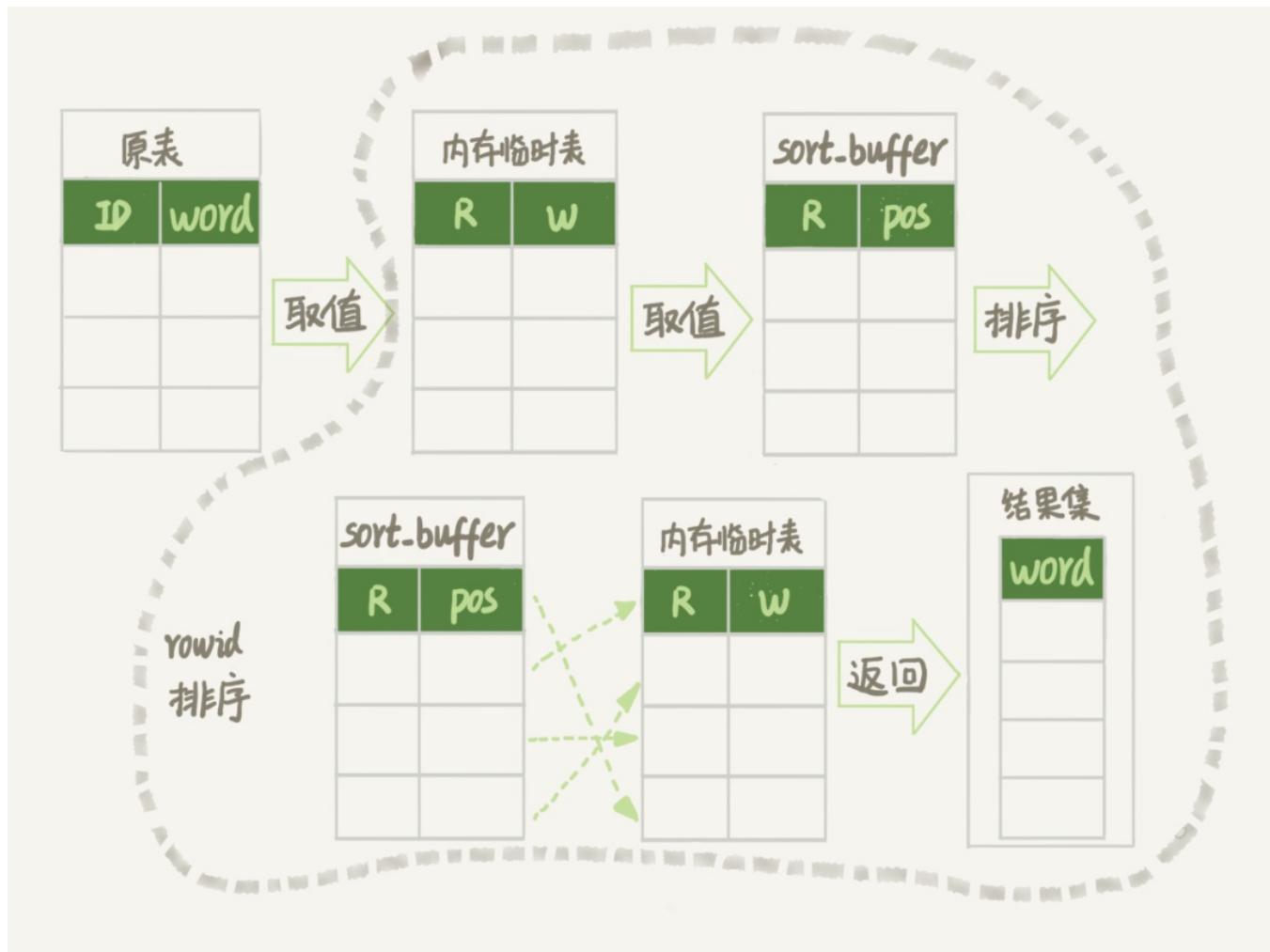


图4 随机排序完整流程图1

图中的**pos**就是位置信息，你可能会觉得奇怪，这里的“位置信息”是个什么概念？在上一篇文章中，我们对InnoDB表排序的时候，明明用的还是**ID**字段。

这时候，我们就要回到一个基本概念：**MySQL**的表是用什么方法来定位“一行数据”的。

在前面[第4](#)和[第5](#)篇介绍索引的文章中，有几位同学问到，如果把一个InnoDB表的主键删掉，是不是就没有主键，就没办法回表了？

其实不是的。如果你创建的表没有主键，或者把一个表的主键删掉了，那么InnoDB会自己生成一个长度为6字节的**rowid**来作为主键。

这也就是排序模式里面，**rowid**名字的来历。实际上它表示的是：每个引擎用来唯一标识数据行的信息。

- 对于有主键的InnoDB表来说，这个**rowid**就是主键**ID**；
- 对于没有主键的InnoDB表来说，这个**rowid**就是由系统生成的；
- MEMORY引擎不是索引组织表。在这个例子里面，你可以认为它就是一个数组。因此，这个**rowid**其实就是数组的下标。

到这里，我来稍微小结一下：**order by rand()**使用了内存临时表，内存临时表排序的时候使用了**rowid**排序方法。

磁盘临时表

那么，是不是所有的临时表都是内存表呢？

其实不是的。**tmp_table_size**这个配置限制了内存临时表的大小，默认值是**16M**。如果临时表大小超过了**tmp_table_size**，那么内存临时表就会转成磁盘临时表。

磁盘临时表使用的引擎默认是**InnoDB**，是由参数**internal_tmp_disk_storage_engine**控制的。

当使用磁盘临时表的时候，对应的就是一个没有显式索引的**InnoDB**表的排序过程。

为了复现这个过程，我把**tmp_table_size**设置成**1024**，把**sort_buffer_size**设置成**32768**，把**max_length_for_sort_data**设置成**16**。

```
set tmp_table_size=1024;
set sort_buffer_size=32768;
set max_length_for_sort_data=16;
/* 打开 optimizer_trace，只对本线程有效 */
SET optimizer_trace='enabled=on';

/* 执行语句 */
select word from words order by rand() limit 3;

/* 查看 OPTIMIZER_TRACE 输出 */
SELECT * FROM `information_schema`.`OPTIMIZER_TRACE`\G
```

```
"filesort_priority_queue_optimization": {
    "limit": 3,
    "rows_estimate": 1213,
    "row_size": 14,
    "memory_available": 32768,
    "chosen": true
},
"filesort_execution": [
],
"filesort_summary": {
    "rows": 4,
    "examined_rows": 10000,
    "number_of_tmp_files": 0,
    "sort_buffer_size": 88,
    "sort_mode": "<sort_key, rowid>"
}
```

图5 OPTIMIZER_TRACE部分结果

然后，我们来看一下这次OPTIMIZER_TRACE的结果。

因为将max_length_for_sort_data设置成16，小于word字段的长度定义，所以我们看到sort_mode里面显示的是rowid排序，这个是符合预期的，参与排序的是随机值R字段和rowid字段组成的行。

这时候你可能心算了一下，发现不对。R字段存放的随机值就8个字节，rowid是6个字节（至于为什么是6字节，就留给你课后思考吧），数据总行数是10000，这样算出来就有140000字节，超过了sort_buffer_size 定义的 32768字节了。但是，number_of_tmp_files的值居然是0，难道不需要用临时文件吗？

这个SQL语句的排序确实没有用到临时文件，采用是MySQL 5.6版本引入的一个新的排序算法，即：优先队列排序算法。接下来，我们就看看为什么没有使用临时文件的算法，也就是归并排序算法，而是采用了优先队列排序算法。

其实，我们现在的SQL语句，只需要取R值最小的3个rowid。但是，如果使用归并排序算法的话，虽然最终也能得到前3个值，但是这个算法结束后，已经将10000行数据都排好序了。

也就是说，后面的9997行也是有序的了。但，我们的查询并不需要这些数据是有序的。所以，想一下就明白了，这浪费了非常多的计算量。

而优先队列算法，就可以精确地只得到三个最小值，执行流程如下：

1. 对于这10000个准备排序的(R,rowid)，先取前三行，构造成一个堆；

(对数据结构印象模糊的同学，可以先设想成这是一个由三个元素组成的数组)

1. 取下一个行($R', rowid'$)，跟当前堆里面最大的 R 比较，如果 R' 小于 R ，把这个($R, rowid$)从堆中去掉，换成($R', rowid'$)；
2. 重复第2步，直到第10000个($R', rowid'$)完成比较。

这里我简单画了一个优先队列排序过程的示意图。

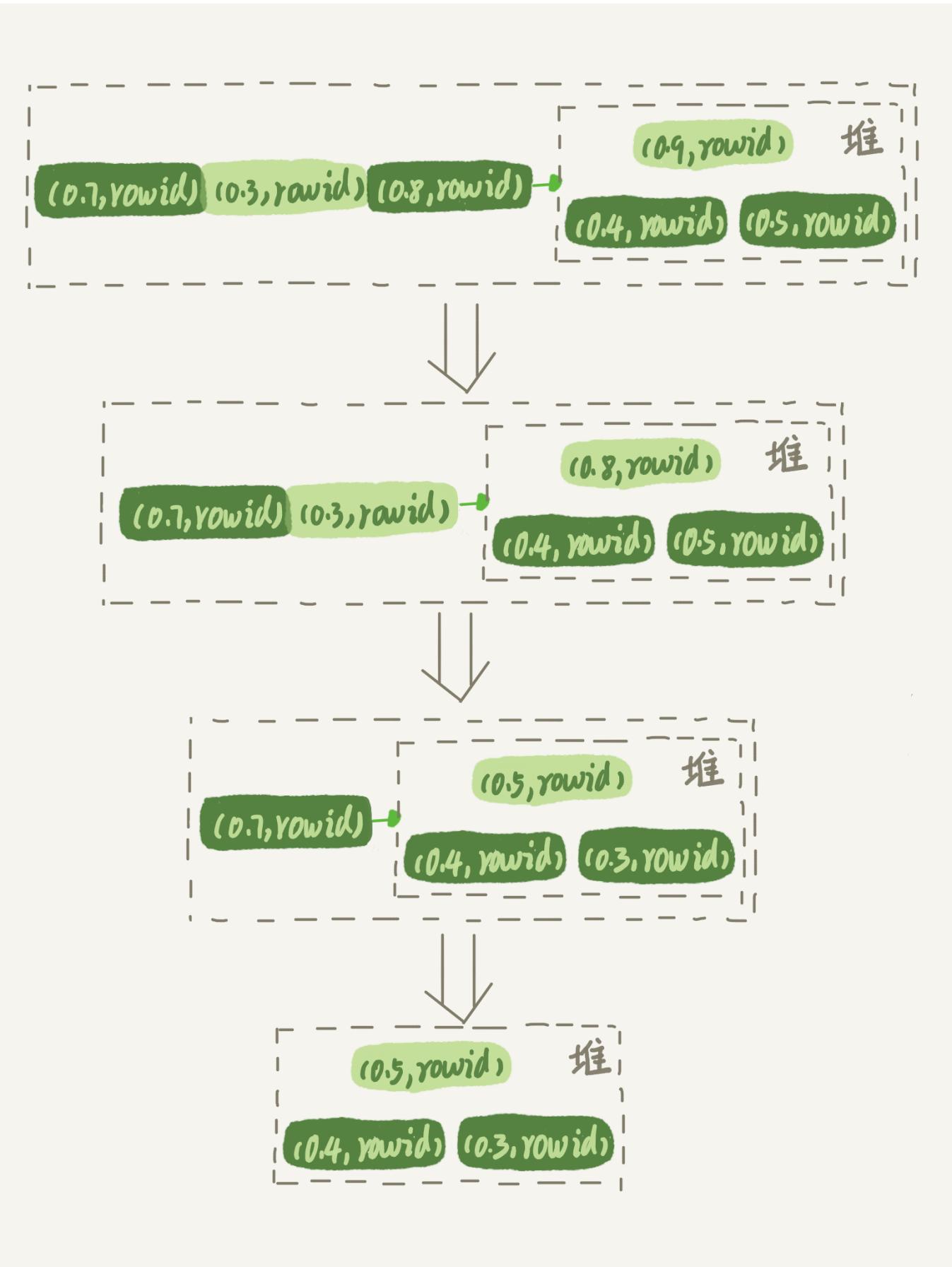


图6 优先队列排序算法示例

图6是模拟6个(R,rowid)行，通过优先队列排序找到最小的三个R值的行的过程。整个排序过程中，为了最快地拿到当前堆的最大值，总是保持最大值在堆顶，因此这是一个最大堆。

图5的OPTIMIZER_TRACE结果中，`filesort_priority_queue_optimization`这个部分的`chosen=true`，就表示使用了优先队列排序算法，这个过程不需要临时文件，因此对应的`number_of_tmp_files`是0。

这个流程结束后，我们构造的堆里面，就是这个10000行里面R值最小的三行。然后，依次把它们的`rowid`拿出来，去临时表里面拿到`word`字段，这个过程就跟上一篇文章的`rowid`排序的过程一样了。

我们再看一下上面一篇文章的SQL查询语句：

```
select city,name,age from t where city='杭州' order by name limit 1000 ;
```

你可能会问，这里也用到了`limit`，为什么没用优先队列排序算法呢？原因是，这条SQL语句是`limit 1000`，如果使用优先队列算法的话，需要维护的堆的大小就是1000行的(`name, rowid`)，超过了我设置的`sort_buffer_size`大小，所以只能使用归并排序算法。

总之，不论是使用哪种类型的临时表，`order by rand()`这种写法都会让计算过程非常复杂，需要大量的扫描行数，因此排序过程的资源消耗也会很大。

再回到我们文章开头的问题，怎么正确地随机排序呢？

随机排序方法

我们先把问题简化一下，如果只随机选择1个`word`值，可以怎么做呢？思路上是这样的：

1. 取得这个表的主键`id`的最大值M和最小值N；
2. 用随机函数生成一个最大值到最小值之间的数 $X = (M-N) * \text{rand}() + N$ ；
3. 取不小于X的第一个ID的行。

我们把这个算法，暂时称作随机算法1。这里，我直接给你贴一下执行语句的序列：

```
mysql> select max(id),min(id) into @M,@N from t ;
set @X= floor((@M-@N+1)*rand() + @N);
select * from t where id >= @X limit 1;
```

这个方法效率很高，因为取`max(id)`和`min(id)`都是不需要扫描索引的，而第三步的`select`也可以用索引快速定位，可以认为就只扫描了3行。但实际上，这个算法本身并不严格满足题目的随机要求，因为ID中间可能有空洞，因此选择不同行的概率不一样，不是真正的随机。

比如你有4个id，分别是1、2、4、5，如果按照上面的方法，那么取到 id=4的这一行的概率是取得其他行概率的两倍。

如果这四行的id分别是1、2、40000、40001呢？这个算法基本就能当bug来看待了。

所以，为了得到严格随机的结果，你可以用下面这个流程：

1. 取得整个表的行数，并记为C。
2. 取得 $Y = \text{floor}(C * \text{rand}())$ 。 floor函数在这里的作用，就是取整数部分。
3. 再用 limit Y,1 取得一行。

我们把这个算法，称为随机算法2。下面这段代码，就是上面流程的执行语句的序列。

```
mysql> select count(*) into @C from t;
set @Y = floor(@C * rand());
set @sql = concat("select * from t limit ", @Y, ",1");
prepare stmt from @sql;
execute stmt;
DEALLOCATE prepare stmt;
```

由于limit 后面的参数不能直接跟变量，所以在上面的代码中使用了prepare+execute的方法。你也可以把拼接SQL语句的方法写在应用程序中，会更简单些。

这个随机算法2，解决了算法1里面明显的概率不均匀问题。

MySQL处理limit Y,1 的做法就是按顺序一个一个地读出来，丢掉前Y个，然后把下一个记录作为返回结果，因此这一步需要扫描Y+1行。再加上，第一步扫描的C行，总共需要扫描C+Y+1行，执行代价比随机算法1的代价要高。

当然，随机算法2跟直接order by rand()比起来，执行代价还是小很多的。

你可能问了，如果按照这个表有10000行来计算的话， $C=10000$ ，要是随机到比较大的Y值，那扫描行数也跟20000差不多了，接近order by rand()的扫描行数，为什么说随机算法2的代价要小很多呢？我就把这个问题留给你去课后思考吧。

现在，我们再看看，如果我们按照随机算法2的思路，要随机取3个word值呢？你可以这么做：

1. 取得整个表的行数，记为C；
2. 根据相同的随机方法得到Y1、Y2、Y3；

3. 再执行三个`limit Y, 1`语句得到三行数据。

我们把这个算法，称作随机算法3。下面这段代码，就是上面流程的执行语句的序列。

```
mysql> select count(*) into @C from t;
set @Y1 = floor(@C * rand());
set @Y2 = floor(@C * rand());
set @Y3 = floor(@C * rand());
select * from t limit @Y1, 1; //在应用代码里面取Y1、Y2、Y3值，拼出SQL后执行
select * from t limit @Y2, 1;
select * from t limit @Y3, 1;
```

小结

今天这篇文章，我是借着随机排序的需求，跟你介绍了MySQL对临时表排序的执行过程。

如果你直接使用`order by rand()`，这个语句需要Using temporary 和 Using filesort，查询的执行代价往往是比较大的。所以，在设计的时候你要尽量避开这种写法。

今天的例子里面，我们不是仅仅在数据库内部解决问题，还会让应用代码配合拼接SQL语句。在实际应用的过程中，比较规范的用法就是：尽量将业务逻辑写在业务代码中，让数据库只做“读写数据”的事情。因此，这类方法的应用还是比较广泛的。

最后，我给你留下一个思考题吧。

上面的随机算法3的总扫描行数是 $C + (Y_1 + 1) + (Y_2 + 1) + (Y_3 + 1)$ ，实际上它还是可以继续优化，来进一步减少扫描行数的。

我的问题是，如果你是这个需求的开发人员，你会怎么做，来减少扫描行数呢？说说你的方案，并说明你的方案需要的扫描行数。

你可以把你的设计和结论写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章最后留给你的问题是，`select * from t where city in ("杭州", "苏州") order by name limit 100;`这个SQL语句是否需要排序？有什么方案可以避免排序？

虽然有`(city, name)`联合索引，对于单个`city`内部，`name`是递增的。但是由于这条SQL语句不是要单独地查一个`city`的值，而是同时查了“杭州”和“苏州”两个城市，因此所有满足条件的`name`就不是递增的了。也就是说，这条SQL语句需要排序。

那怎么避免排序呢？

这里，我们要用到(**city,name**)联合索引的特性，把这一条语句拆成两条语句，执行流程如下：

1. 执行**select * from t where city="杭州" order by name limit 100;** 这个语句是不需要排序的，客户端用一个长度为100的内存数组**A**保存结果。
2. 执行**select * from t where city="苏州" order by name limit 100;** 用相同的方法，假设结果被存进了内存数组**B**。
3. 现在**A**和**B**是两个有序数组，然后你可以用归并排序的思想，得到**name**最小的前100值，就是我们需要的结果了。

如果把这条SQL语句里“**limit 100**”改成“**limit 10000,100**”的话，处理方式其实也差不多，即：要把上面的两条语句改成写：

```
select * from t where city="杭州" order by name limit 10100;
```

和

```
select * from t where city="苏州" order by name limit 10100.
```

这时候数据量较大，可以同时起两个连接一行行读结果，用归并排序算法拿到这两个结果集里，按顺序取第**10001~10100**的**name**值，就是需要的结果了。

当然这个方案有一个明显的损失，就是从数据库返回给客户端的数据量变大了。

所以，如果数据的单行比较大的话，可以考虑把这两条SQL语句改成下面这种写法：

```
select id,name from t where city="杭州" order by name limit 10100;
```

和

```
select id,name from t where city="苏州" order by name limit 10100.
```

然后，再用归并排序的方法取得按**name**顺序第**10001~10100**的**name**、**id**的值，然后拿着这100个**id**到数据库中去查出所有记录。

上面这些方法，需要你根据性能需求和开发的复杂度做出权衡。

评论区留言点赞板：

评论区很多同学都提到不能排序，说明各位对索引的存储都理解对了。

@峰 同学提到了归并排序，是我们这个问题解法的核心思想；

@老杨同志 的回答中提到了“从业务上砍掉功能”，这个也确实是在业务设计中可以考虑的一个方向；

@某、人 帮忙回答了@发条橙子同学的问题，尤其是对问题一的回答，非常精彩。

The image shows a promotional banner for a MySQL course. On the left, there's a logo for '极客时间' (Geek Time) with a orange play button icon. The main title 'MySQL 实战 45 讲' is displayed prominently in large, dark font. Below it, a subtitle reads '从原理到实战，丁奇带你搞懂 MySQL' (From principle to practice, Ding Qi will guide you to understand MySQL). To the right of the text, there's a portrait of a man with glasses and short hair, wearing a black shirt, with his arms crossed. At the bottom left, the teacher's name '林晓斌' is listed along with his nickname '丁奇' and title '前阿里资深技术专家' (Former senior technical expert at Alibaba). A call-to-action at the bottom says '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。' (New version upgrade: Click 'Ask friends to read', get 10 free reads, and invite to subscribe for cash rewards).

精选留言



老杨同志

21

对应单词这种总量不是很多的数据，第一感觉应该装jdk缓存或者redis缓存。由于需要随机访问，数组比较好。假如一个单词平均10个字节， $10 * 10000$ ，不到1M就装下了。

如果一定要用数据库来做，老师的方案1比较好，空洞的问题，如果单词库不变，可以在上线前整理数据，把空洞处理掉。比如：原来单词存在A表，新建B表，执行 `insert into B(word) select word from A`. B的id是自增的，就会生成连续的主键。当然如果A表写比较频繁，且数据量较大，业务上禁用这种写法，RR的隔离级别会锁A表

2018-12-21

| 作者回复

重新整理表这个思路很赞

看得出你是业务经验很丰富啊，这几次问题，对底层实现和业务功能的平衡，考虑点很不错

2018-12-21



雪中鼠[悠闲]

4

如果按照业务需求，随机取三个，数据库还在设计阶段，可以增加一个主键字段，用来记录每行记录的**rowid**，这样一万行，那就是连续的一万，然后随机，用该随机**rowid**回表查询该行记录

2018-12-21

| 作者回复

这个也是个好方法，就是确保连续，可以快速的得到C和几个偏移量

2018-12-21



吴宇晨

16

我觉得可以按Y排个序，第一条取完，拿到对应id，然后有一条语句就是**where id大于xxx, limit y2-y1, 1**

2018-12-21

| 作者回复

抓住了关键点

2018-12-21



慧鑫coming

10

又到周五了，开心

2018-12-21



HuaMax

7

假设Y1, Y2, Y3是由小到大的三个数，则可以优化成这样，这样扫描行数为Y3

```
id1 = select * from t limit @Y1, 1;  
id2= select * from t where id > id1 limit @Y2-@Y1, 1;  
select * from t where id > id2 limit @Y3 - @Y2, 1;
```

2018-12-21

| 作者回复

||

2018-12-21



freesia

4

从上一讲到这一讲，我发现老师在处理问题时，提出的方法就不再是单纯依靠MySQL解决，因为可能会耗费很多资源，而是把问题分担一部分到客户端，比如客户端拿到数据后再排序，或者客户端产生随机数再到MySQL中去查询。

2018-12-23

| 作者回复

嗯嗯，MySQL 的代码和业务代码都是代码配合起来用

2018-12-23



李皮皮皮皮皮

4

我经常在文中看到多个事务的执行时序。线下做实验的时候，是怎么保证能按这个时序执行呢？

2018-12-21

| 作者回复

开两个窗口，按顺序执行命令哦

2018-12-21



岁月安然

2

为什么随机算法2比order by rand()的代价小很多？

因为随机算法2进行limit获取数据的时候是根据主键排序获取的，主键天然索引排序。获取到第9999条的数据也远比order by rand()方法的组成临时表R字段排序再获取rowid代价小的多。

2018-12-21

| 作者回复

对的，

你是第一个回答正文中间问题的！

2018-12-21



倪大人

2

课后题可以在随机出Y1、Y2、Y3后，算出Ymax、Ymin

再用 select id from t limit Ymin, (Ymax - Ymin);

得到id集后算出Y1、Y2、Y3对应的三个id

最后 select * from t where id in (id1, id2, id3)

这样扫描的行数应该是C+Ymax+3

2018-12-21

| 作者回复

漂亮

2018-12-21



董航

2

堆结构，大顶树，小顶树！！！

2018-12-21



王飞洋

2

归并排序，优先队列，算法无处不在。

2018-12-21

| 作者回复

要说算法还是隔壁王老师讲的专业，这里咱们就只追求MySQL里面用到的，能给大家讲明白就行了！

2018-12-21



某、人

1

今天这个问题我的理解转换成sql是：

```
mysql> select count(*) into @C from t1;
set @Y = floor(@C * rand());
```

```
set @Y1 = floor(@C * rand());
set @Y2 = floor(@C * rand());
select LEAST(@Y,@Y1,@Y2) into @Y4;
select GREATEST(@Y,@Y1,@Y2) into @Y6;
select floor((@Y6+@Y4)/2) into @Y5;
set @sql = concat("select id into @id from t1 limit ", @Y4, ",1");
set @sql1 = concat("select id into @id1 from t1 where id>@id limit ", @Y5-@Y4, ",1");
set @sql2 = concat("select id into @id2 from t1 where id>@id1 limit ", @Y6-@Y5, ",1");
prepare stmt from @sql;
prepare stmt1 from @sql1;
prepare stmt2 from @sql2;
execute stmt;
execute stmt1;
execute stmt2;
DEALLOCATE prepare stmt;
DEALLOCATE prepare stmt1;
DEALLOCATE prepare stmt2;
select * from t1 where id in (@id,@id1,@id2);
```

感觉mysql不太适合处理随机数的问题,稍稍有点复杂。

不过这两节课收获很多,对order by排序理解又深入不少,原来堆排序是放limit m,m行如果比sort_buffer占用空间小,则先把m行放进数据集里,然后在把表里的数据一行一行取出来做比较。得出的结果,在根据MRR回表取数据。

老师,我有一个问题:

堆排序,如果比较的值是相等的情况下,会不会替换在sort_buffer里? 我感觉是不会,如果不会才能解释得通排序值相等,id不等的情况,不管是大顶堆还是小顶堆,得到的结果集都是id相对更小的

2018-12-23



路过

1

老师, 我为快速执行存储过程。把参数位置为:

```
innodb_flush_log_at_trx_commit=2
sync_binlog=0
```

执行马上就结束了。否则要等很久。请教老师, 上面修改后, 数据和log还没有真正刷到磁盘。
请问我在哪里可以看到相关的信息。

使用show engine innodb status\G 看到:

```
0 pending log flushes, 0 pending chkp writes
20197 log i/o's done, 0.00 log i/o's/second
```

谢谢!

2018-12-22

| 作者回复

确实没地方看

2018-12-22



风动草

1

老师好！您说的在建二级索引的过程中，是把主键取出来构造二级索引，而且要读全表，这个读全表意思是不是，读了主键，就意味着主键的叶子节点也一起读出来了？

2018-12-22

| 作者回复

是的

2018-12-22



无眠

1

一直比较疑惑什么情况下会产生临时表Using temporary，希望老师指点下

2018-12-21

| 作者回复

查询需要临时表，比如我们这个例子里，需要临时表来放rand()结果

2018-12-21



银太@巨益科技

1

请教下老师：

表A有sku和warehouse两个字段组成的唯一索引,udx_sku_warehouse，高并发下容易死锁

执行的语句： update A set quantity=quantity+1 where sku=xx and warehouse=xx

查看死锁的日志：两个事务都在等待udx_sku_warehouse的X锁，但两个事务修改的并不是同一条记录，不是很明白，可以讲解一下吗？多谢

*** (1) TRANSACTION:

TRANSACTION 466841895, ACTIVE 0.021 sec starting index read

mysql tables in use 1, locked 1

LOCK WAIT 11 lock struct(s), heap size 2936, 9 row lock(s), undo log entries 11

LOCK BLOCKING MySQL thread id: 1927379 block 1895984

MySQL thread id 1895984, OS thread handle 0x2b2ffed85700, query id 783954740 10.27.8.222

oms updating

UPDATE oms_stock

SET quantity = quantity + -1

WHERE sku_id = 13978218638755841

AND virtual_warehouse_id = 13867758969455616

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 297 page no 89 n bits 424 index `udx_sku_id_warehouse_id` of tabl

e `oms_biz`.`oms_stock` trx id 466841895 lock_mode X locks rec but not gap waiting

Record lock, heap no 18 PHYSICAL RECORD: n_fields 3; compact format; info bits 0

*** (2) TRANSACTION:

TRANSACTION 466841901, ACTIVE 0.015 sec starting index read

mysql tables in use 1, locked 1

11 lock struct(s), heap size 2936, 8 row lock(s), undo log entries 9

MySQL thread id 1927379, OS thread handle 0x2b2f97440700, query id 783954758 10.27.8.222

oms updating

UPDATE oms_stock

```
SET quantity = quantity + -1
WHERE sku_id = 1809040003028
AND virtual_warehouse_id = 13867758969455616
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 297 page no 89 n bits 424 index `udx_sku_id_warehouse_id` of table `oms_biz`.`oms_stock` trx id 466841901 lock_mode X locks rec but not gap
Record lock, heap no 18 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
```

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:

```
RECORD LOCKS space id 297 page no 74 n bits 400 index `udx_sku_id_warehouse_id` of table `oms_biz`.`oms_stock` trx id 466841901 lock_mode X locks rec but not gap waiting
Record lock, heap no 12 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
```

2018-12-21

| 作者回复

你一个事务里面是不是不止一个这样的update 语句?

2018-12-21



penelopewu

1

运行老师给的存储过程特别慢，怎么排查原因呢，mysql版本是8.0.13

2018-12-21

| 作者回复

把innodb_flush_at_trx_commit设置成2, sync_binlog设置成1000看看

2018-12-21



往事随风，顺其自然

1

临时表设置参数单位是k还是m?

2018-12-21

| 作者回复

字节

2018-12-21



奋斗心

0

20000行是指：扫描10000行到内存临时表，还有10000行是随机排序吗

2019-02-02

| 作者回复

第一个10000是扫描原表，第二个10000是扫描内存表；

排序过程本身是不增加扫描行数的

2019-02-03



阿狸爱JAVA

0

感觉老师的思路很宽广，就像一个大宝藏，方案一不行还有方案二，方案二不行还有方案三，

并且每个方案都能给出具体的性能比较与证据，而自己自能顺着老师的思路还能明白，可是一旦扩展开来，便大脑一片空白

2019-01-31

| 作者回复

加油慢慢来哈~~~

2019-01-31

17 | 如何正确地显示随机消息？

2018-12-21 林晓斌



我在上一篇文章，为你讲解完**order by**语句的几种执行模式后，就想到了之前一个做英语学习App的朋友碰到过的一个性能问题。今天这篇文章，我就从这个性能问题说起，和你说说MySQL中的另外一种排序需求，希望能够加深你对MySQL排序逻辑的理解。

这个英语学习App首页有一个随机显示单词的功能，也就是根据每个用户的级别有一个单词表，然后这个用户每次访问首页的时候，都会随机滚动显示三个单词。他们发现随着单词表变大，选单词这个逻辑变得越来越慢，甚至影响到了首页的打开速度。

现在，如果让你来设计这个SQL语句，你会怎么写呢？

为了便于理解，我对这个例子进行了简化：去掉每个级别的用户都有一个对应的单词表这个逻辑，直接就是从一个单词表中随机选出三个单词。这个表的建表语句和初始数据的命令如下：

```

mysql> CREATE TABLE `words` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `word` varchar(64) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;

delimiter ;;
create procedure idata()
begin
    declare i int;
    set i=0;
    while i<10000 do
        insert into words(word) values(concat(char(97+(i div 1000)), char(97+(i % 1000 div 100))), set i=i+1;
    end while;
end;;
delimiter ;

call idata();

```

为了便于量化说明，我在这个表里面插入了10000行记录。接下来，我们就一起看看要随机选择3个单词，有什么方法实现，存在什么问题以及如何改进。

内存临时表

首先，你会想到用`order by rand()`来实现这个逻辑。

```
mysql> select word from words order by rand() limit 3;
```

这个语句的意思很直白，随机排序取前3个。虽然这个SQL语句写法很简单，但执行流程却有点复杂的。

我们先用`explain`命令来看看这个语句的执行情况。

```
mysql> explain select word from words order by rand() limit 3;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | words | NULL       | ALL   | NULL          | NULL | NULL    | NULL | 9980 | 100.00  | Using temporary; Using filesort |
```

图1 使用explain命令查看语句的执行情况

Extra字段显示Using temporary, 表示的是需要使用临时表; Using filesort, 表示的是需要执行排序操作。

因此这个Extra的意思就是，需要临时表，并且需要在临时表上排序。

这里，你可以先回顾一下[上一篇文章](#)中全字段排序和rowid排序的内容。我把上一篇文章的两个流程图贴过来，方便你复习。

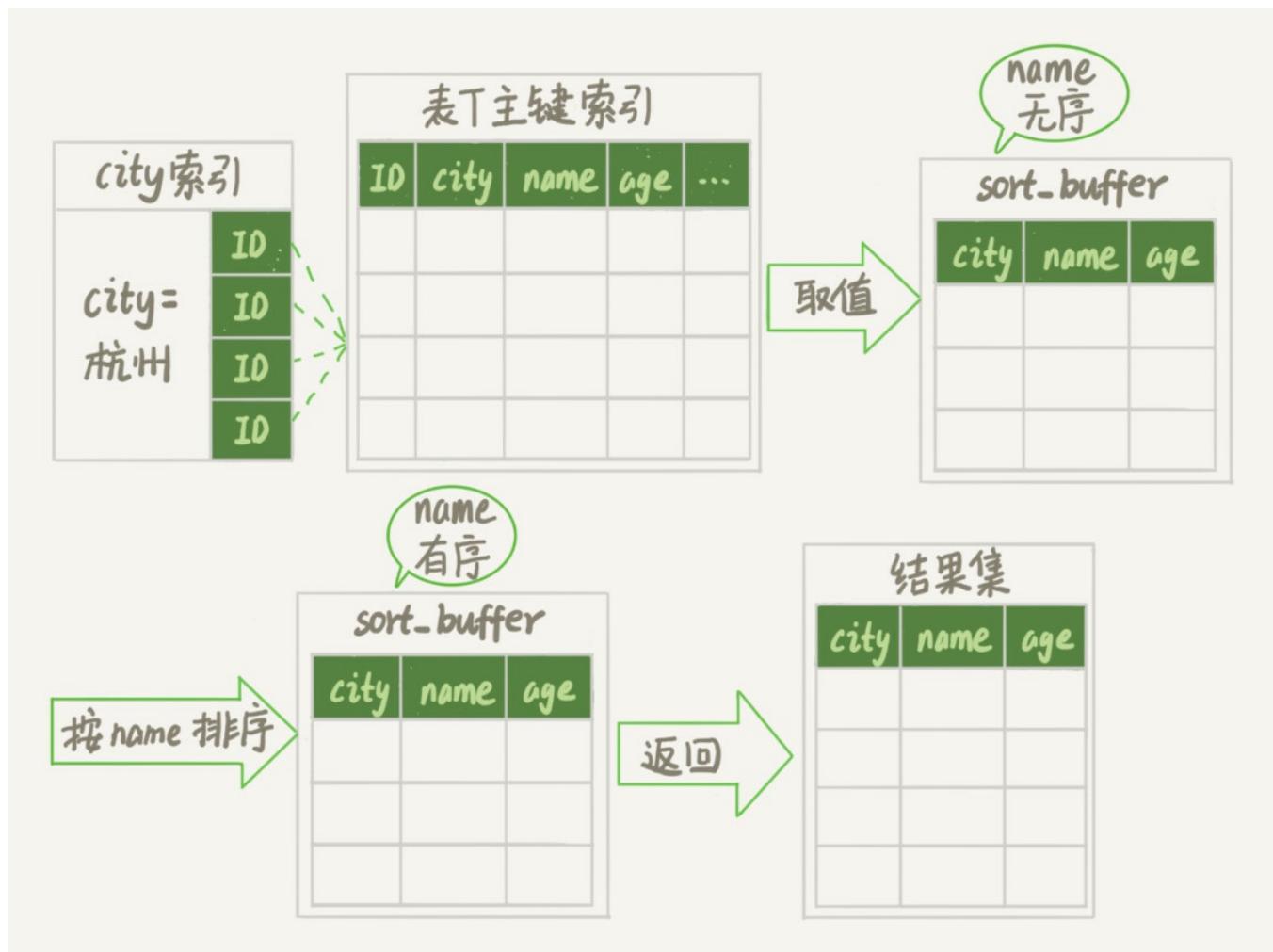


图2 全字段排序

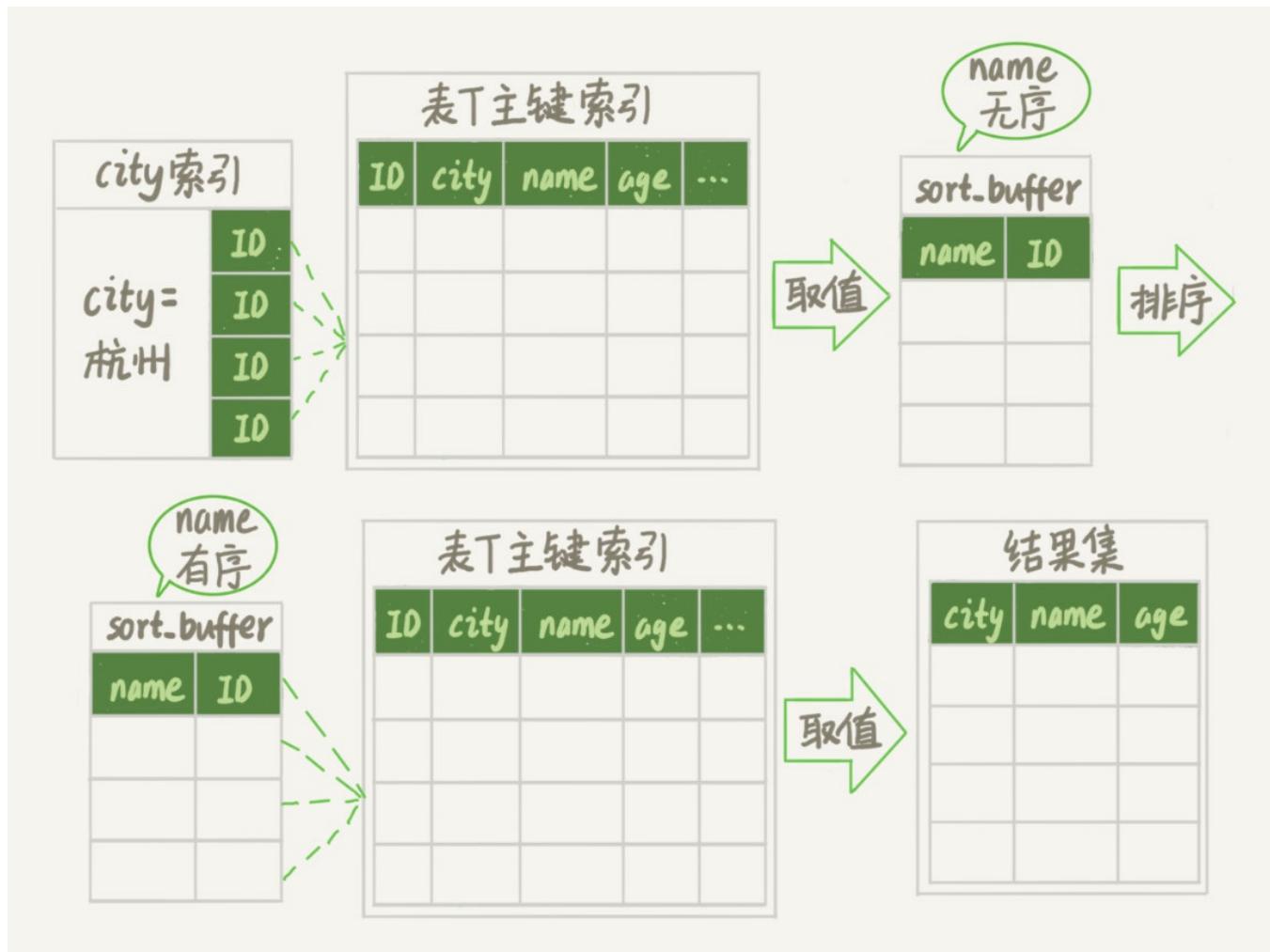


图3 rowid排序

然后，我再问你一个问题，你觉得对于临时内存表的排序来说，它会选择哪一种算法呢？回顾一下上一篇文章的一个结论：对于**InnoDB表**来说，执行全字段排序会减少磁盘访问，因此会被优先选择。

我强调了“**InnoDB表**”，你肯定想到了，对于内存表，回表过程只是简单地根据数据行的位置，直接访问内存得到数据，根本不会导致多访问磁盘。优化器没有了这一层顾虑，那么它会优先考虑的，就是用于排序的行越少越好了，所以，**MySQL**这时就会选择**rowid**排序。

理解了这个算法选择的逻辑，我们再来看看语句的执行流程。同时，通过今天的这个例子，我们来尝试分析一下语句的扫描行数。

这条语句的执行流程是这样的：

1. 创建一个临时表。这个临时表使用的是**memory**引擎，表里有两个字段，第一个字段是**double**类型，为了后面描述方便，记为字段R，第二个字段是**varchar(64)**类型，记为字段W。并且，这个表没有建索引。
2. 从**words**表中，按主键顺序取出所有的**word**值。对于每一个**word**值，调用**rand()**函数生成一个大于0小于1的随机小数，并把这个随机小数和**word**分别存入临时表的R和W字段中，到

此，扫描行数是**10000**。

3. 现在临时表有**10000**行数据了，接下来你要在这个没有索引的内存临时表上，按照字段**R**排序。
4. 初始化 **sort_buffer**。**sort_buffer**中有两个字段，一个是**double**类型，另一个是整型。
5. 从内存临时表中一行一行地取出**R**值和位置信息（我后面会和你解释这里为什么是“位置信息”），分别存入**sort_buffer**中的两个字段里。这个过程要对内存临时表做全表扫描，此时扫描行数增加**10000**，变成了**20000**。
6. 在**sort_buffer**中根据**R**的值进行排序。注意，这个过程没有涉及到表操作，所以不会增加扫描行数。
7. 排序完成后，取出前三个结果的位置信息，依次到内存临时表中取出**word**值，返回给客户端。这个过程中，访问了表的三行数据，总扫描行数变成了**20003**。

接下来，我们通过慢查询日志（**slow log**）来验证一下我们分析得到的扫描行数是否正确。

```
# Query_time: 0.900376 Lock_time: 0.000347 Rows_sent: 3 Rows_examined: 20003
SET timestamp=1541402277;
select word from words order by rand() limit 3;
```

其中，**Rows_examined: 20003**就表示这个语句执行过程中扫描了**20003**行，也就验证了我们分析得出的结论。

这里插一句题外话，在平时学习概念的过程中，你可以经常这样做，先通过原理分析算出扫描行数，然后再通过查看慢查询日志，来验证自己的结论。我自己就是经常这么做，这个过程很有趣，分析对了开心，分析错了但是弄清楚了也很开心。

现在，我来把完整的排序执行流程图画出来。

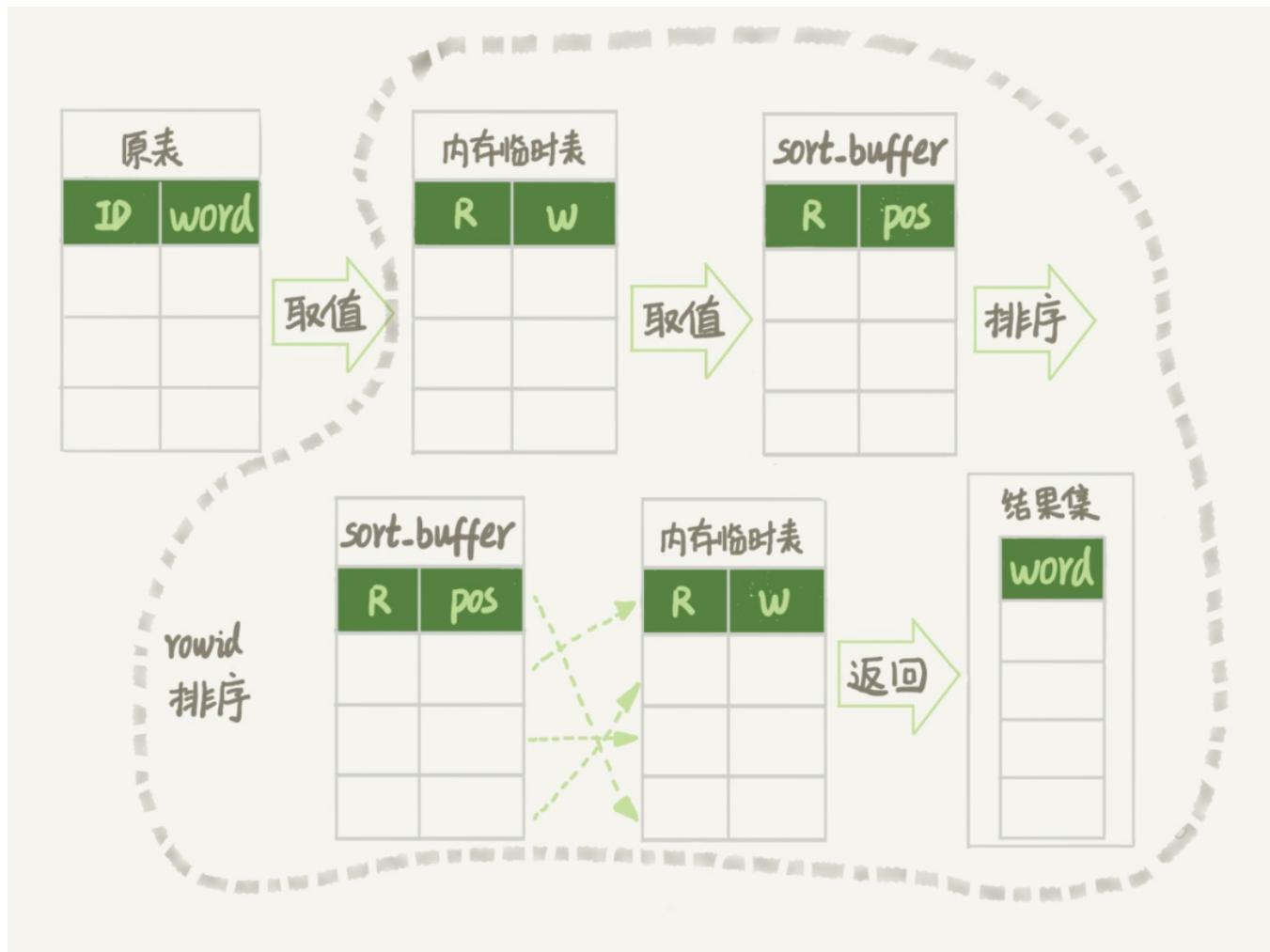


图4 随机排序完整流程图1

图中的**pos**就是位置信息，你可能会觉得奇怪，这里的“位置信息”是个什么概念？在上一篇文章中，我们对InnoDB表排序的时候，明明用的还是**ID**字段。

这时候，我们就要回到一个基本概念：**MySQL**的表是用什么方法来定位“一行数据”的。

在前面[第4](#)和[第5](#)篇介绍索引的文章中，有几位同学问到，如果把一个InnoDB表的主键删掉，是不是就没有主键，就没办法回表了？

其实不是的。如果你创建的表没有主键，或者把一个表的主键删掉了，那么InnoDB会自己生成一个长度为6字节的**rowid**来作为主键。

这也就是排序模式里面，**rowid**名字的来历。实际上它表示的是：每个引擎用来唯一标识数据行的信息。

- 对于有主键的InnoDB表来说，这个**rowid**就是主键**ID**；
- 对于没有主键的InnoDB表来说，这个**rowid**就是由系统生成的；
- MEMORY引擎不是索引组织表。在这个例子里面，你可以认为它就是一个数组。因此，这个**rowid**其实就是数组的下标。

到这里，我来稍微小结一下：**order by rand()**使用了内存临时表，内存临时表排序的时候使用了**rowid**排序方法。

磁盘临时表

那么，是不是所有的临时表都是内存表呢？

其实不是的。**tmp_table_size**这个配置限制了内存临时表的大小，默认值是**16M**。如果临时表大小超过了**tmp_table_size**，那么内存临时表就会转成磁盘临时表。

磁盘临时表使用的引擎默认是**InnoDB**，是由参数**internal_tmp_disk_storage_engine**控制的。

当使用磁盘临时表的时候，对应的就是一个没有显式索引的**InnoDB**表的排序过程。

为了复现这个过程，我把**tmp_table_size**设置成**1024**，把**sort_buffer_size**设置成**32768**，把**max_length_for_sort_data**设置成**16**。

```
set tmp_table_size=1024;
set sort_buffer_size=32768;
set max_length_for_sort_data=16;
/* 打开 optimizer_trace，只对本线程有效 */
SET optimizer_trace='enabled=on';

/* 执行语句 */
select word from words order by rand() limit 3;

/* 查看 OPTIMIZER_TRACE 输出 */
SELECT * FROM `information_schema`.`OPTIMIZER_TRACE`\G
```

```
"filesort_priority_queue_optimization": {
    "limit": 3,
    "rows_estimate": 1213,
    "row_size": 14,
    "memory_available": 32768,
    "chosen": true
},
"filesort_execution": [
],
"filesort_summary": {
    "rows": 4,
    "examined_rows": 10000,
    "number_of_tmp_files": 0,
    "sort_buffer_size": 88,
    "sort_mode": "<sort_key, rowid>"
}
```

图5 OPTIMIZER_TRACE部分结果

然后，我们来看一下这次OPTIMIZER_TRACE的结果。

因为将max_length_for_sort_data设置成16，小于word字段的长度定义，所以我们看到sort_mode里面显示的是rowid排序，这个是符合预期的，参与排序的是随机值R字段和rowid字段组成的行。

这时候你可能心算了一下，发现不对。R字段存放的随机值就8个字节，rowid是6个字节（至于为什么是6字节，就留给你课后思考吧），数据总行数是10000，这样算出来就有140000字节，超过了sort_buffer_size 定义的 32768字节了。但是，number_of_tmp_files的值居然是0，难道不需要用临时文件吗？

这个SQL语句的排序确实没有用到临时文件，采用是MySQL 5.6版本引入的一个新的排序算法，即：优先队列排序算法。接下来，我们就看看为什么没有使用临时文件的算法，也就是归并排序算法，而是采用了优先队列排序算法。

其实，我们现在的SQL语句，只需要取R值最小的3个rowid。但是，如果使用归并排序算法的话，虽然最终也能得到前3个值，但是这个算法结束后，已经将10000行数据都排好序了。

也就是说，后面的9997行也是有序的了。但，我们的查询并不需要这些数据是有序的。所以，想一下就明白了，这浪费了非常多的计算量。

而优先队列算法，就可以精确地只得到三个最小值，执行流程如下：

1. 对于这10000个准备排序的(R,rowid)，先取前三行，构造成一个堆；

(对数据结构印象模糊的同学，可以先设想成这是一个由三个元素组成的数组)

1. 取下一个行($R', rowid'$)，跟当前堆里面最大的 R 比较，如果 R' 小于 R ，把这个($R, rowid$)从堆中去掉，换成($R', rowid'$)；
2. 重复第2步，直到第10000个($R', rowid'$)完成比较。

这里我简单画了一个优先队列排序过程的示意图。

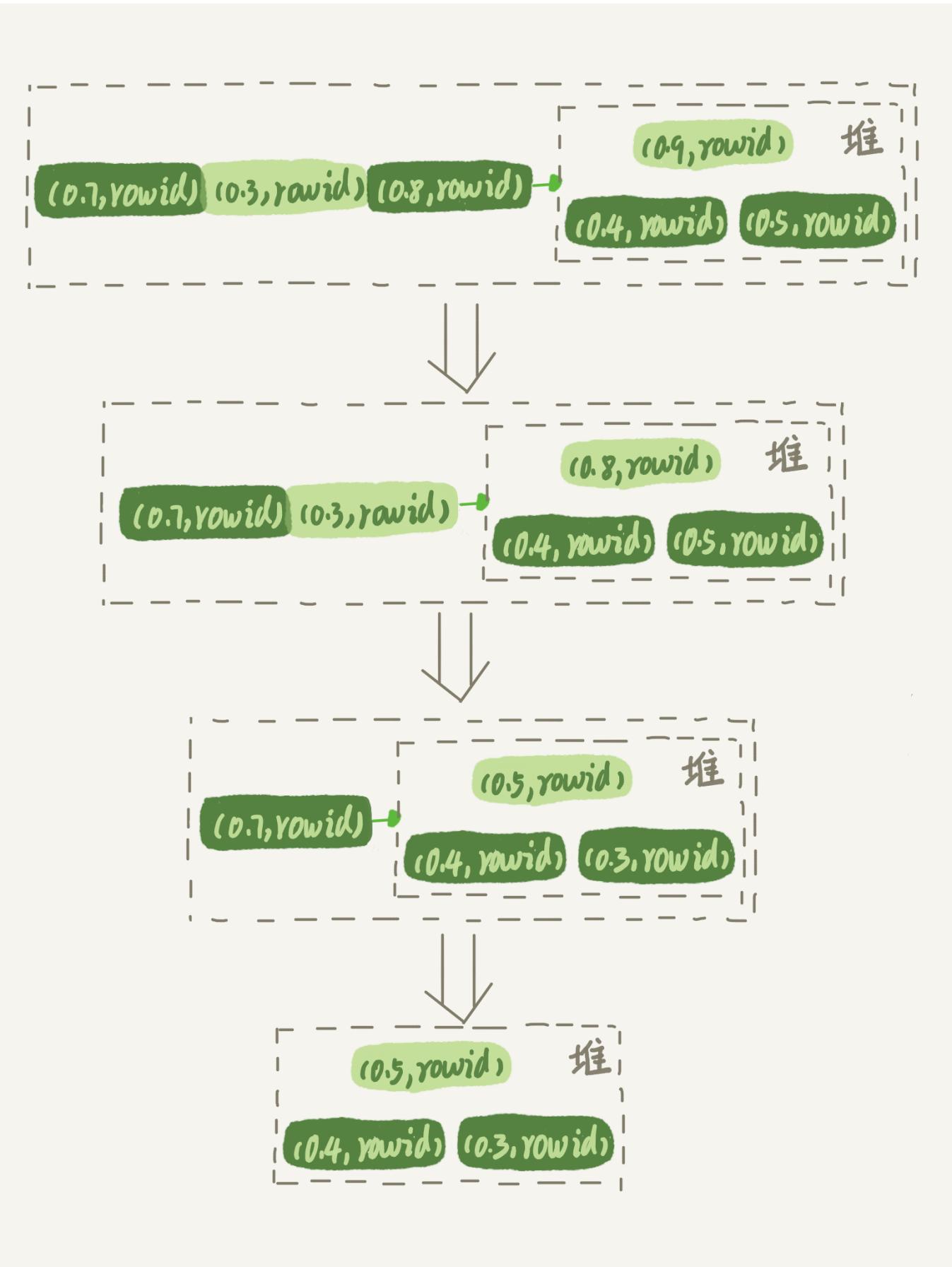


图6 优先队列排序算法示例

图6是模拟6个(R,rowid)行，通过优先队列排序找到最小的三个R值的行的过程。整个排序过程中，为了最快地拿到当前堆的最大值，总是保持最大值在堆顶，因此这是一个最大堆。

图5的OPTIMIZER_TRACE结果中，`filesort_priority_queue_optimization`这个部分的`chosen=true`，就表示使用了优先队列排序算法，这个过程不需要临时文件，因此对应的`number_of_tmp_files`是0。

这个流程结束后，我们构造的堆里面，就是这个10000行里面R值最小的三行。然后，依次把它们的`rowid`拿出来，去临时表里面拿到`word`字段，这个过程就跟上一篇文章的`rowid`排序的过程一样了。

我们再看一下上面一篇文章的SQL查询语句：

```
select city,name,age from t where city='杭州' order by name limit 1000 ;
```

你可能会问，这里也用到了`limit`，为什么没用优先队列排序算法呢？原因是，这条SQL语句是`limit 1000`，如果使用优先队列算法的话，需要维护的堆的大小就是1000行的(`name, rowid`)，超过了我设置的`sort_buffer_size`大小，所以只能使用归并排序算法。

总之，不论是使用哪种类型的临时表，`order by rand()`这种写法都会让计算过程非常复杂，需要大量的扫描行数，因此排序过程的资源消耗也会很大。

再回到我们文章开头的问题，怎么正确地随机排序呢？

随机排序方法

我们先把问题简化一下，如果只随机选择1个`word`值，可以怎么做呢？思路上是这样的：

1. 取得这个表的主键`id`的最大值M和最小值N；
2. 用随机函数生成一个最大值到最小值之间的数 $X = (M-N) * \text{rand}() + N$ ；
3. 取不小于X的第一个ID的行。

我们把这个算法，暂时称作随机算法1。这里，我直接给你贴一下执行语句的序列：

```
mysql> select max(id),min(id) into @M,@N from t ;
set @X= floor((@M-@N+1)*rand() + @N);
select * from t where id >= @X limit 1;
```

这个方法效率很高，因为取`max(id)`和`min(id)`都是不需要扫描索引的，而第三步的`select`也可以用索引快速定位，可以认为就只扫描了3行。但实际上，这个算法本身并不严格满足题目的随机要求，因为ID中间可能有空洞，因此选择不同行的概率不一样，不是真正的随机。

比如你有4个id，分别是1、2、4、5，如果按照上面的方法，那么取到 id=4的这一行的概率是取得其他行概率的两倍。

如果这四行的id分别是1、2、40000、40001呢？这个算法基本就能当bug来看待了。

所以，为了得到严格随机的结果，你可以用下面这个流程：

1. 取得整个表的行数，并记为C。
2. 取得 $Y = \text{floor}(C * \text{rand}())$ 。 floor函数在这里的作用，就是取整数部分。
3. 再用 limit Y,1 取得一行。

我们把这个算法，称为随机算法2。下面这段代码，就是上面流程的执行语句的序列。

```
mysql> select count(*) into @C from t;
set @Y = floor(@C * rand());
set @sql = concat("select * from t limit ", @Y, ",1");
prepare stmt from @sql;
execute stmt;
DEALLOCATE prepare stmt;
```

由于limit 后面的参数不能直接跟变量，所以在上面的代码中使用了prepare+execute的方法。你也可以把拼接SQL语句的方法写在应用程序中，会更简单些。

这个随机算法2，解决了算法1里面明显的概率不均匀问题。

MySQL处理limit Y,1 的做法就是按顺序一个一个地读出来，丢掉前Y个，然后把下一个记录作为返回结果，因此这一步需要扫描Y+1行。再加上，第一步扫描的C行，总共需要扫描C+Y+1行，执行代价比随机算法1的代价要高。

当然，随机算法2跟直接order by rand()比起来，执行代价还是小很多的。

你可能问了，如果按照这个表有10000行来计算的话，C=10000，要是随机到比较大的Y值，那扫描行数也跟20000差不多了，接近order by rand()的扫描行数，为什么说随机算法2的代价要小很多呢？我就把你这个问题留给你去课后思考吧。

现在，我们再看看，如果我们按照随机算法2的思路，要随机取3个word值呢？你可以这么做：

1. 取得整个表的行数，记为C；
2. 根据相同的随机方法得到Y1、Y2、Y3；

3. 再执行三个`limit Y, 1`语句得到三行数据。

我们把这个算法，称作随机算法3。下面这段代码，就是上面流程的执行语句的序列。

```
mysql> select count(*) into @C from t;
set @Y1 = floor(@C * rand());
set @Y2 = floor(@C * rand());
set @Y3 = floor(@C * rand());
select * from t limit @Y1, 1; //在应用代码里面取Y1、Y2、Y3值，拼出SQL后执行
select * from t limit @Y2, 1;
select * from t limit @Y3, 1;
```

小结

今天这篇文章，我是借着随机排序的需求，跟你介绍了MySQL对临时表排序的执行过程。

如果你直接使用`order by rand()`，这个语句需要Using temporary 和 Using filesort，查询的执行代价往往是比较大的。所以，在设计的时候你要尽量避开这种写法。

今天的例子里面，我们不是仅仅在数据库内部解决问题，还会让应用代码配合拼接SQL语句。在实际应用的过程中，比较规范的用法就是：尽量将业务逻辑写在业务代码中，让数据库只做“读写数据”的事情。因此，这类方法的应用还是比较广泛的。

最后，我给你留下一个思考题吧。

上面的随机算法3的总扫描行数是 $C + (Y_1 + 1) + (Y_2 + 1) + (Y_3 + 1)$ ，实际上它还是可以继续优化，来进一步减少扫描行数的。

我的问题是，如果你是这个需求的开发人员，你会怎么做，来减少扫描行数呢？说说你的方案，并说明你的方案需要的扫描行数。

你可以把你的设计和结论写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章最后留给你的问题是，`select * from t where city in ("杭州", "苏州") order by name limit 100;`这个SQL语句是否需要排序？有什么方案可以避免排序？

虽然有`(city, name)`联合索引，对于单个`city`内部，`name`是递增的。但是由于这条SQL语句不是要单独地查一个`city`的值，而是同时查了“杭州”和“苏州”两个城市，因此所有满足条件的`name`就不是递增的了。也就是说，这条SQL语句需要排序。

那怎么避免排序呢？

这里，我们要用到(**city,name**)联合索引的特性，把这一条语句拆成两条语句，执行流程如下：

1. 执行**select * from t where city="杭州" order by name limit 100;** 这个语句是不需要排序的，客户端用一个长度为100的内存数组**A**保存结果。
2. 执行**select * from t where city="苏州" order by name limit 100;** 用相同的方法，假设结果被存进了内存数组**B**。
3. 现在**A**和**B**是两个有序数组，然后你可以用归并排序的思想，得到**name**最小的前100值，就是我们需要的结果了。

如果把这条SQL语句里“**limit 100**”改成“**limit 10000,100**”的话，处理方式其实也差不多，即：要把上面的两条语句改成写：

```
select * from t where city="杭州" order by name limit 10100;
```

和

```
select * from t where city="苏州" order by name limit 10100.
```

这时候数据量较大，可以同时起两个连接一行行读结果，用归并排序算法拿到这两个结果集里，按顺序取第**10001~10100**的**name**值，就是需要的结果了。

当然这个方案有一个明显的损失，就是从数据库返回给客户端的数据量变大了。

所以，如果数据的单行比较大的话，可以考虑把这两条SQL语句改成下面这种写法：

```
select id,name from t where city="杭州" order by name limit 10100;
```

和

```
select id,name from t where city="苏州" order by name limit 10100.
```

然后，再用归并排序的方法取得按**name**顺序第**10001~10100**的**name**、**id**的值，然后拿着这100个**id**到数据库中去查出所有记录。

上面这些方法，需要你根据性能需求和开发的复杂度做出权衡。

评论区留言点赞板：

评论区很多同学都提到不能排序，说明各位对索引的存储都理解对了。

@峰 同学提到了归并排序，是我们这个问题解法的核心思想；

@老杨同志 的回答中提到了“从业务上砍掉功能”，这个也确实是在业务设计中可以考虑的一个方向；

@某、人 帮忙回答了@发条橙子同学的问题，尤其是对问题一的回答，非常精彩。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Lin Xiaobin, a man with short dark hair and glasses, wearing a black button-down shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45 讲' in large, bold, dark font, with the subtitle '从原理到实战，丁奇带你搞懂 MySQL' below it. Above the title is the '极客时间' logo. On the far left, there is a section with the name '林晓斌' and the text '网名丁奇' and '前阿里资深技术专家'. At the bottom, there is a call-to-action text: '新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。'.

18 | 为什么这些SQL语句逻辑相同，性能却差异巨大？

2018-12-24 林晓斌



在MySQL中，有很多看上去逻辑相同，但性能却差异巨大的SQL语句。对这些语句使用不当的话，就会不经意间导致整个数据库的压力变大。

我今天挑选了三个这样的案例和你分享。希望再遇到相似的问题时，你可以做到举一反三、快速解决问题。

案例一：条件字段函数操作

假设你现在维护了一个交易系统，其中交易记录表`tradelog`包含交易流水号（`tradeid`）、交易员`id`（`operator`）、交易时间（`t_modified`）等字段。为了便于描述，我们先忽略其他字段。这个表的建表语句如下：

```
mysql> CREATE TABLE `tradelog` (
    `id` int(11) NOT NULL,
    `tradeid` varchar(32) DEFAULT NULL,
    `operator` int(11) DEFAULT NULL,
    `t_modified` datetime DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY `tradeid` (`tradeid`),
    KEY `t_modified` (`t_modified`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

假设，现在已经记录了从2016年初到2018年底的所有数据，运营部门有一个需求是，要统计发生在所有年份中7月份的交易记录总数。这个逻辑看上去并不复杂，你的SQL语句可能会这么写：

```
mysql> select count(*) from tradelog where month(t_modified)=7;
```

由于t_modified字段上有索引，于是你就很放心地在生产库中执行了这条语句，但却发现执行了特别久，才返回了结果。

如果你问DBA同事为什么会出现这样的情况，他大概会告诉你：如果对字段做了函数计算，就用不上索引了，这是MySQL的规定。

现在你已经学过了InnoDB的索引结构了，可以再追问一句为什么？为什么条件是where t_modified='2018-7-1'的时候可以用上索引，而改成where month(t_modified)=7的时候就不行了？

下面是这个t_modified索引的示意图。方框上面的数字就是month()函数对应的值。

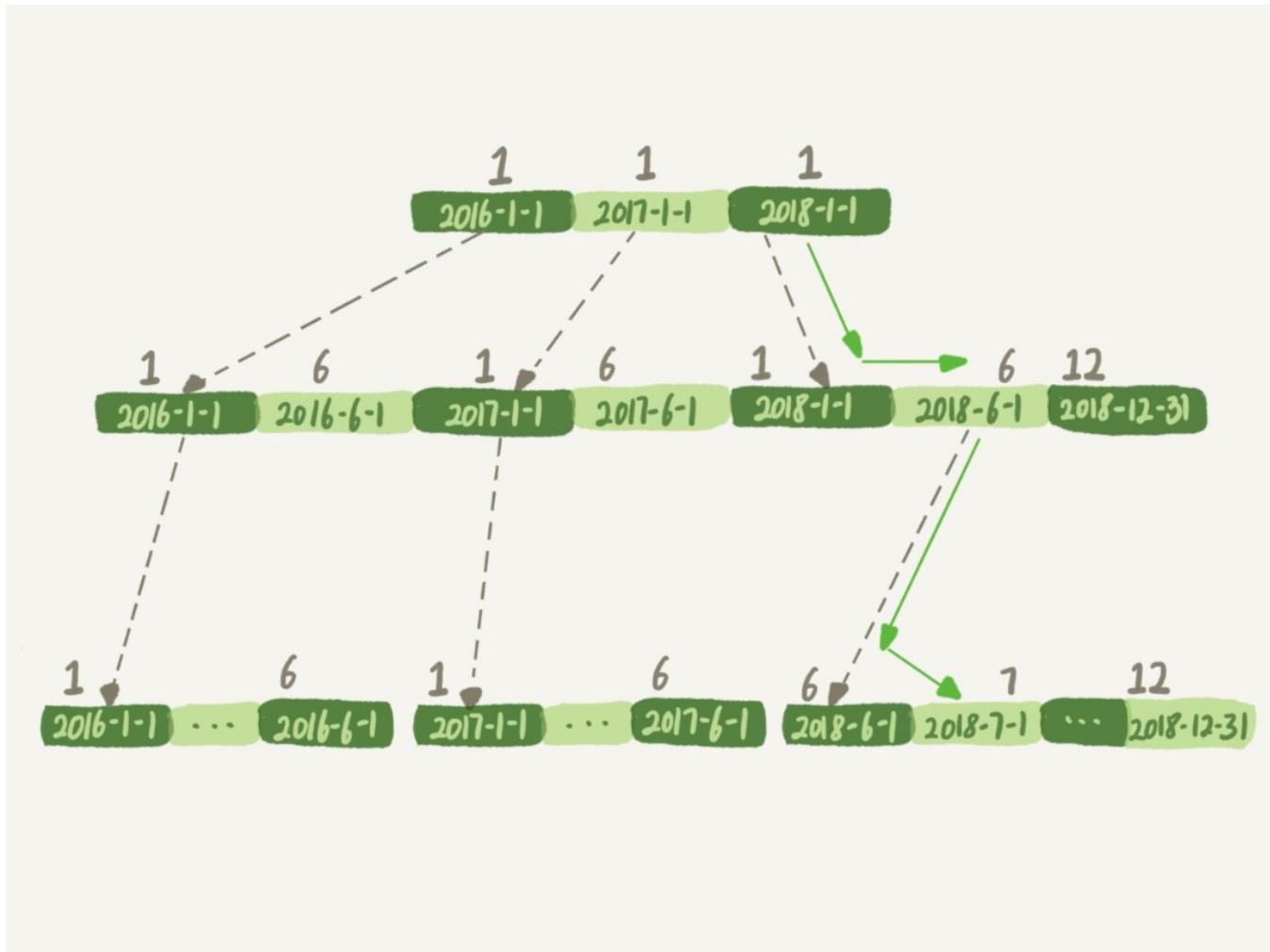


图1 t_modified索引示意图

如果你的SQL语句条件用的是`where t_modified='2018-7-1'`的话，引擎就会按照上面绿色箭头的路线，快速定位到`t_modified='2018-7-1'`需要的结果。

实际上，B+树提供的这个快速定位能力，来源于同一层兄弟节点的有序性。

但是，如果计算`month()`函数的话，你会看到传入7的时候，在树的第一层就不知道该怎么办了。

也就是说，对索引字段做函数操作，可能会破坏索引值的有序性，因此优化器就决定放弃走树搜索功能。

需要注意的是，优化器并不是要放弃使用这个索引。

在这个例子里，放弃了树搜索功能，优化器可以选择遍历主键索引，也可以选择遍历索引`t_modified`，优化器对比索引大小后发现，索引`t_modified`更小，遍历这个索引比遍历主键索引来得更快。因此最终还是会选择索引`t_modified`。

接下来，我们使用`explain`命令，查看一下这条SQL语句的执行结果。

```
mysql> explain select count(*) from tradelog where month(t_modified)=7;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tradelog | NULL | index | NULL | t_modified | 6 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 100335 | 100.00 | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图2 explain 结果

`key="t_modified"`表示的是，使用了`t_modified`这个索引；我在测试表数据中插入了10万行数据，`rows=100335`，说明这条语句扫描了整个索引的所有值；`Extra`字段的`Using index`，表示的是使用了覆盖索引。

也就是说，由于在`t_modified`字段加了`month()`函数操作，导致了全索引扫描。为了能够用上索引的快速定位能力，我们就要把SQL语句改成基于字段本身的范围查询。按照下面这个写法，优化器就能按照我们预期的，用上`t_modified`索引的快速定位能力了。

```
mysql> select count(*) from tradelog where
-> (t_modified >= '2016-7-1' and t_modified<'2016-8-1') or
-> (t_modified >= '2017-7-1' and t_modified<'2017-8-1') or
-> (t_modified >= '2018-7-1' and t_modified<'2018-8-1');
```

当然，如果你的系统上线时间更早，或者后面又插入了之后年份的数据的话，你就需要再把其他年份补齐。

到这里我给你说明了，由于加了`month()`函数操作，MySQL无法再使用索引快速定位功能，而只能使用全索引扫描。

不过优化器在这个问题上确实有“偷懒”行为，即使是对于不改变有序性的函数，也不会考虑使用索引。比如，对于`select * from tradelog where id + 1 = 10000`这个SQL语句，这个加1操作并不会改变有序性，但是MySQL优化器还是不能用`id`索引快速定位到9999这一行。所以，需要你在写SQL语句的时候，手动改写成`where id = 10000 - 1`才可以。

案例二：隐式类型转换

接下来我再跟你说一说，另一个经常让程序员掉坑里的例子。

我们一起看一下这条SQL语句：

```
mysql> select * from tradelog where tradeid=110717;
```

交易编号`tradeid`这个字段上，本来就有索引，但是`explain`的结果却显示，这条语句需要走全表扫描。你可能也发现了，`tradeid`的字段类型是`varchar(32)`，而输入的参数却是整型，所以需要做类型转换。

那么，现在这里就有两个问题：

1. 数据类型转换的规则是什么？
2. 为什么有数据类型转换，就需要走全索引扫描？

先来看第一个问题，你可能会说，数据库里面类型这么多，这种数据类型转换规则更多，我记不住，应该怎么办呢？

这里有一个简单的方法，看 `select "10" > 9` 的结果：

1. 如果规则是“将字符串转成数字”，那么就是做数字比较，结果应该是1；
2. 如果规则是“将数字转成字符串”，那么就是做字符串比较，结果应该是0。

验证结果如图3所示。

```
mysql> select "10" > 9;
+-----+
| "10" > 9 |
+-----+
|      1    |
+-----+
```

图3 MySQL中字符串和数字转换的效果示意图

从图中可知，`select "10" > 9` 返回的是1，所以你就能确认MySQL里的转换规则了：在MySQL中，字符串和数字做比较的话，是将字符串转换成数字。

这时，你再看这个全表扫描的语句：

```
mysql> select * from tradelog where tradeid=110717;
```

就知道对于优化器来说，这个语句相当于：

```
mysql> select * from tradelog where CAST(tradid AS signed int) = 110717;
```

也就是说，这条语句触发了我们上面说到的规则：对索引字段做函数操作，优化器会放弃走树搜索功能。

现在，我留给你一个小问题，`id`的类型是`int`，如果执行下面这个语句，是否会导致全表扫描呢？

```
select * from tradelog where id="83126";
```

你可以先自己分析一下，再到数据库里面去验证确认。

接下来，我们再来看一个稍微复杂点的例子。

案例三：隐式字符编码转换

假设系统里还有另外一个表trade_detail，用于记录交易的操作细节。为了便于量化分析和复现，我往交易日志表tradelog和交易详情表trade_detail这两个表里插入一些数据。

```
mysql> CREATE TABLE `trade_detail` (
    `id` int(11) NOT NULL,
    `tradeid` varchar(32) DEFAULT NULL,
    `trade_step` int(11) DEFAULT NULL, /*操作步骤*/
    `step_info` varchar(32) DEFAULT NULL, /*步骤信息*/
    PRIMARY KEY (`id`),
    KEY `tradeid` (`tradeid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert into tradelog values(1, 'aaaaaaaa', 1000, now());
insert into tradelog values(2, 'aaaaaaaab', 1000, now());
insert into tradelog values(3, 'aaaaaaaac', 1000, now());

insert into trade_detail values(1, 'aaaaaaaa', 1, 'add');
insert into trade_detail values(2, 'aaaaaaaa', 2, 'update');
insert into trade_detail values(3, 'aaaaaaaa', 3, 'commit');
insert into trade_detail values(4, 'aaaaaaaab', 1, 'add');
insert into trade_detail values(5, 'aaaaaaaab', 2, 'update');
insert into trade_detail values(6, 'aaaaaaaab', 3, 'update again');
insert into trade_detail values(7, 'aaaaaaaab', 4, 'commit');
insert into trade_detail values(8, 'aaaaaaaac', 1, 'add');
insert into trade_detail values(9, 'aaaaaaaac', 2, 'update');
insert into trade_detail values(10, 'aaaaaaaac', 3, 'update again');
insert into trade_detail values(11, 'aaaaaaaac', 4, 'commit');
```

这时候，如果要查询id=2的交易的所有操作步骤信息，SQL语句可以这么写：

```
mysql> select d.* from tradelog l, trade_detail d where d.tradeid=l.tradeid and l.id=2; /*语句
```

mysql> explain select d.* from tradelog l , trade_detail d where d.tradeid=l.tradeid and l.id=2;												
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	l	NULL	const	PRIMARY,tradeid	PRIMARY	4	const	1	100.00	NULL	
1	SIMPLE	d	NULL	ALL	NULL	NULL	NULL	NULL	11	100.00	Using where	

图4 语句Q1的explain结果

我们一起来看下这个结果：

1. 第一行显示优化器会先在交易记录表tradelog上查到id=2的行，这个步骤用上了主键索引，rows=1表示只扫描一行；
2. 第二行key=NULL，表示没有用上交易详情表trade_detail上的tradeid索引，进行了全表扫描。

在这个执行计划里，是从tradelog表中取tradeid字段，再去trade_detail表里查询匹配字段。因此，我们把tradelog称为驱动表，把trade_detail称为被驱动表，把tradeid称为关联字段。

接下来，我们看下这个explain结果表示的执行流程：

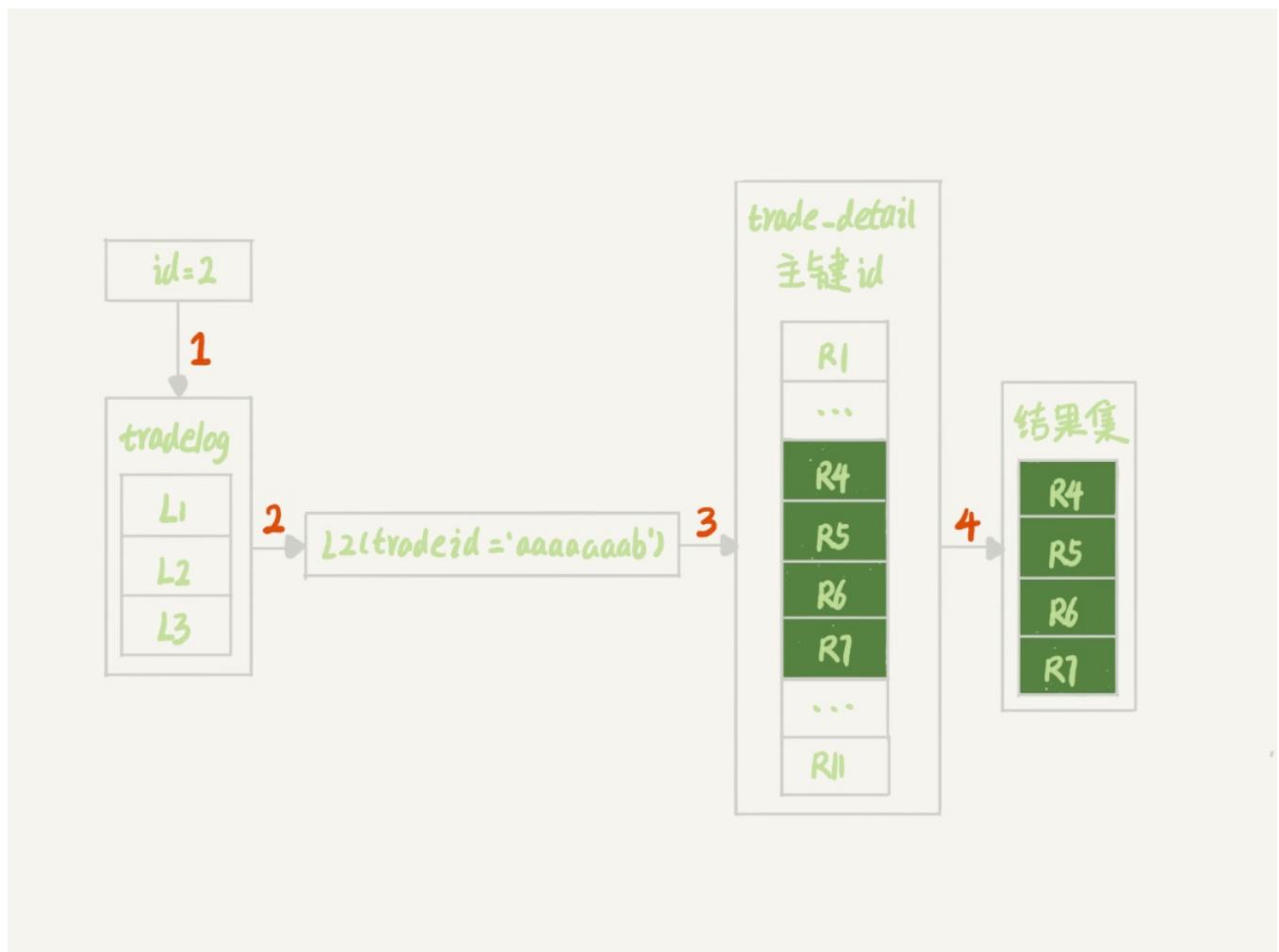


图5 语句Q1的执行过程

图中：

- 第1步，是根据id在tradelog表里找到L2这一行；
- 第2步，是从L2中取出tradeid字段的值；
- 第3步，是根据tradeid值到trade_detail表中查找条件匹配的行。`explain`的结果里面第二行的`key=NULL`表示的就是，这个过程是通过遍历主键索引的方式，一个一个地判断tradeid的值是否匹配。

进行到这里，你会发现第3步不符合我们的预期。因为表trade_detail里tradeid字段上是有索引的，我们本来是希望通过使用tradeid索引能够快速定位到等值的行。但，这里并没有。

如果你去问DBA同学，他们可能会告诉你，因为这两个表的字符集不同，一个是utf8，一个是utf8mb4，所以做表连接查询的时候用不上关联字段的索引。这个回答，也是通常你搜索这个问题时会得到的答案。

但是你应该再追问一下，为什么字符集不同就用不上索引呢？

我们说问题是出在执行步骤的第3步，如果单独把这一步改成SQL语句的话，那就是：

```
mysql> select * from trade_detail where tradeid=$L2.tradeid.value;
```

其中，\$L2.tradeid.value的字符集是utf8mb4。

参照前面的两个例子，你肯定就想到了，字符集utf8mb4是utf8的超集，所以当这两个类型的字符串在做比较的时候，MySQL内部的操作是，先把utf8字符串转成utf8mb4字符集，再做比较。

这个设定很好理解，utf8mb4是utf8的超集。类似地，在程序设计语言里面，做自动类型转换的时候，为了避免数据在转换过程中由于截断导致数据错误，也都是“按数据长度增加的方向”进行转换的。

因此，在执行上面这个语句的时候，需要将被驱动数据表里的字段一个个地转换成utf8mb4，再跟L2做比较。

也就是说，实际上这个语句等同于下面这个写法：

```
select * from trade_detail where CONVERT(tradeid USING utf8mb4)=$L2.tradeid.value;
```

CONVERT()函数，在这里的意思是把输入的字符串转成utf8mb4字符集。

这就再次触发了我们上面说到的原则：对索引字段做函数操作，优化器会放弃走树搜索功能。

到这里，你终于明确了，字符集不同只是条件之一，连接过程中要求在被驱动表的索引字段上加函数操作，是直接导致对被驱动表做全表扫描的原因。

作为对比验证，我给你提另外一个需求，“查找trade_detail表里id=4的操作，对应的操作者是谁”，再来看下这个语句和它的执行计划。

```
mysql> select l.operator from tradelog l , trade_detail d where d.tradeid=l.tradeid and d.id=4;
+-----+
| l.operator |
+-----+
|          |
+-----+
1 row in set (0.00 sec)

mysql> explain select l.operator from tradelog l , trade_detail d where d.tradeid=l.tradeid and d.id=4;
+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+
| 1 | SIMPLE     | d    | NULL       | const | PRIMARY      | PRIMARY | 4      | const | 1   | 100.00 | NULL    |
| 1 | SIMPLE     | l    | NULL       | ref   | tradeid      | tradeid | 131    | const | 1   | 100.00 | NULL    |
+-----+
2 rows in set, 1 warning (0.00 sec)
```

图6 explain 结果

这个语句里trade_detail表成了驱动表，但是explain结果的第二行显示，这次的查询操作用上了被驱动表tradelog里的索引(tradeid)，扫描行数是1。

这也是两个tradeid字段的join操作，为什么这次能用上被驱动表的tradeid索引呢？我们来分析一下。

假设驱动表trade_detail里id=4的行记为R4，那么在连接的时候（图5的第3步），被驱动表tradelog上执行的就是类似这样的SQL语句：

```
select operator from tradelog where traideid =$R4.tradeid.value;
```

这时候\$R4.tradeid.value的字符集是utf8，按照字符集转换规则，要转成utf8mb4，所以这个过程就被改写成：

```
select operator from tradelog where traideid =CONVERT($R4.tradeid.value USING utf8mb4);
```

你看，这里的CONVERT函数是加在输入参数上的，这样就可以用上被驱动表的traideid索引。

理解了原理以后，就可以用来指导操作了。如果要优化语句

```
select d.* from tradelog l, trade_detail d where d.tradeid=l.tradeid and l.id=2;
```

的执行过程，有两种做法：

- 比较常见的优化方法是，把trade_detail表上的tradeid字段的字符集也改成utf8mb4，这样就

没有字符集转换的问题了。

```
alter table trade_detail modify tradeid varchar(32) CHARACTER SET utf8mb4 default null;
```

- 如果能够修改字段的字符集的话，是最好不过了。但如果数据量比较大，或者业务上暂时不能做这个DDL的话，那就只能采用修改SQL语句的方法了。

```
mysql> select d.* from tradelog l , trade_detail d where d.tradeid=CONVERT(l.tradeid USING utf8) and l.id=2;
mysql> explain select d.* from tradelog l , trade_detail d where d.tradeid=CONVERT(l.tradeid USING utf8) and l.id=2;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | l | NULL | const | PRIMARY | PRIMARY | 4 | const | 1 | 100.00 | NULL |
| 1 | SIMPLE | d | NULL | ref | tradeid | tradeid | 99 | const | 4 | 100.00 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

图7 SQL语句优化后的explain结果

这里，我主动把l.tradeid转成utf8，就避免了被驱动表上的字符编码转换，从explain结果可以看到，这次索引走对了。

小结

今天我给你举了三个例子，其实是在说同一件事儿，即：对索引字段做函数操作，可能会破坏索引值的有序性，因此优化器就决定放弃走树搜索功能。

第二个例子是隐式类型转换，第三个例子是隐式字符编码转换，它们都跟第一个例子一样，因为要求在索引字段上做函数操作而导致了全索引扫描。

MySQL的优化器确实有“偷懒”的嫌疑，即使简单地把where id+1=1000改写成where id=1000-1就能够用上索引快速查找，也不会主动做这个语句重写。

因此，每次你的业务代码升级时，把可能出现的、新的SQL语句explain一下，是一个很好的习惯。

最后，又到了思考题时间。

今天我留给你的课后问题是，你遇到过别的、类似今天我们提到的性能问题吗？你认为原因是什么，又是怎么解决的呢？

你可以把你经历和分析写在留言区里，我会在下一篇文章的末尾选取有趣的评论跟大家一起分享和分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上篇文章的最后，留给你的问题是：我们文章中最后一个方案是，通过三次`limit Y,1`来得到需要的数据，你觉得有没有进一步的优化方法。

这里我给出一种方法，取Y1、Y2和Y3里面最大的一个数，记为M，最小的一个数记为N，然后执行下面这条SQL语句：

```
mysql> select * from t limit N, M-N+1;
```

再加上取整个表总行数的C行，这个方案的扫描行数总共只需要C+M+1行。

当然也可以先取回id值，在应用中确定了三个id值以后，再执行三次`where id=X`的语句也是可以的。@倪大人同学在评论区就提到了这个方法。

这次评论区出现了很多很棒的留言：

@老杨同志 提出了重新整理的方法、@雪中鼠[悠闲] 提到了用rowid的方法，是类似的思路，就是让表里面保存一个无空洞的自增值，这样就可以用我们的随机算法1来实现；
@吴宇晨 提到了拿到第一个值以后，用id迭代往下找的方案，利用了主键索引的有序性。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Lin Xiaobin, a man with glasses and a black shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45 讲' in large, bold, dark font, with the subtitle '从原理到实战，丁奇带你搞懂 MySQL' below it. On the far left, there's a logo for '极客时间'. At the bottom left, it says '林晓斌 网名丁奇 前阿里资深技术专家'. A call-to-action at the bottom right encourages users to upgrade to a new version and invite friends to read it for free.

极客时间

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌 网名丁奇
前阿里资深技术专家

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。

18 | 为什么这些SQL语句逻辑相同，性能却差异巨大？

2018-12-24 林晓斌



在MySQL中，有很多看上去逻辑相同，但性能却差异巨大的SQL语句。对这些语句使用不当的话，就会不经意间导致整个数据库的压力变大。

我今天挑选了三个这样的案例和你分享。希望再遇到相似的问题时，你可以做到举一反三、快速解决问题。

案例一：条件字段函数操作

假设你现在维护了一个交易系统，其中交易记录表`tradelog`包含交易流水号（`tradeid`）、交易员`id`（`operator`）、交易时间（`t_modified`）等字段。为了便于描述，我们先忽略其他字段。这个表的建表语句如下：

```
mysql> CREATE TABLE `tradelog` (
    `id` int(11) NOT NULL,
    `tradeid` varchar(32) DEFAULT NULL,
    `operator` int(11) DEFAULT NULL,
    `t_modified` datetime DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY `tradeid` (`tradeid`),
    KEY `t_modified` (`t_modified`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

假设，现在已经记录了从2016年初到2018年底的所有数据，运营部门有一个需求是，要统计发生在所有年份中7月份的交易记录总数。这个逻辑看上去并不复杂，你的SQL语句可能会这么写：

```
mysql> select count(*) from tradelog where month(t_modified)=7;
```

由于t_modified字段上有索引，于是你就很放心地在生产库中执行了这条语句，但却发现执行了特别久，才返回了结果。

如果你问DBA同事为什么会出现这样的情况，他大概会告诉你：如果对字段做了函数计算，就用不上索引了，这是MySQL的规定。

现在你已经学过了InnoDB的索引结构了，可以再追问一句为什么？为什么条件是where t_modified='2018-7-1'的时候可以用上索引，而改成where month(t_modified)=7的时候就不行了？

下面是这个t_modified索引的示意图。方框上面的数字就是month()函数对应的值。

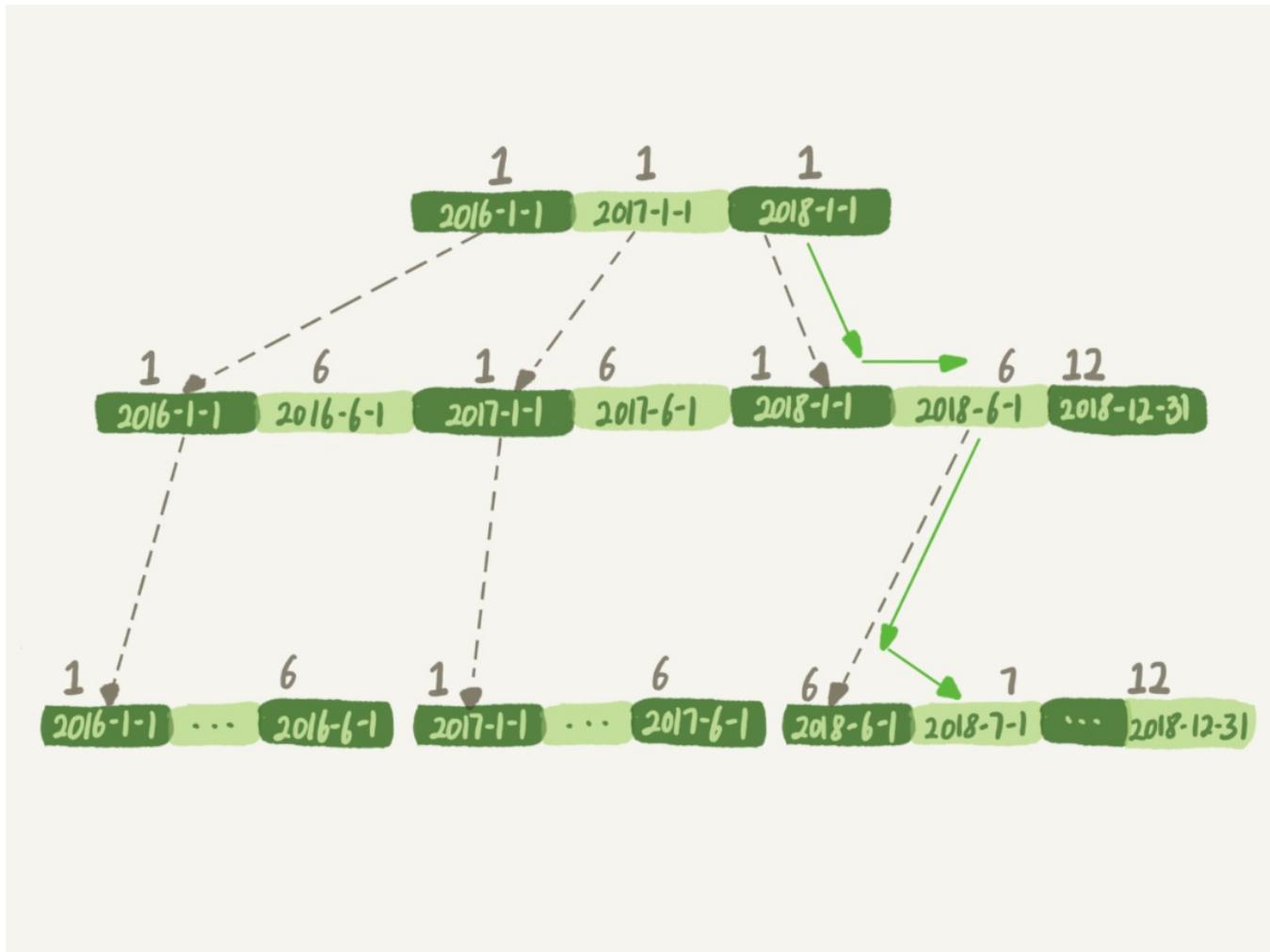


图1 t_modified索引示意图

如果你的SQL语句条件用的是`where t_modified='2018-7-1'`的话，引擎就会按照上面绿色箭头的路线，快速定位到`t_modified='2018-7-1'`需要的结果。

实际上，B+树提供的这个快速定位能力，来源于同一层兄弟节点的有序性。

但是，如果计算`month()`函数的话，你会看到传入7的时候，在树的第一层就不知道该怎么办了。

也就是说，对索引字段做函数操作，可能会破坏索引值的有序性，因此优化器就决定放弃走树搜索功能。

需要注意的是，优化器并不是要放弃使用这个索引。

在这个例子里，放弃了树搜索功能，优化器可以选择遍历主键索引，也可以选择遍历索引`t_modified`，优化器对比索引大小后发现，索引`t_modified`更小，遍历这个索引比遍历主键索引来得更快。因此最终还是会选择索引`t_modified`。

接下来，我们使用`explain`命令，查看一下这条SQL语句的执行结果。

```
mysql> explain select count(*) from tradelog where month(t_modified)=7;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tradelog | NULL | index | NULL | t_modified | 6 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 100335 | 100.00 | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图2 explain 结果

`key="t_modified"`表示的是，使用了`t_modified`这个索引；我在测试表数据中插入了10万行数据，`rows=100335`，说明这条语句扫描了整个索引的所有值；`Extra`字段的`Using index`，表示的是使用了覆盖索引。

也就是说，由于在`t_modified`字段加了`month()`函数操作，导致了全索引扫描。为了能够用上索引的快速定位能力，我们就要把SQL语句改成基于字段本身的范围查询。按照下面这个写法，优化器就能按照我们预期的，用上`t_modified`索引的快速定位能力了。

```
mysql> select count(*) from tradelog where
-> (t_modified >= '2016-7-1' and t_modified<'2016-8-1') or
-> (t_modified >= '2017-7-1' and t_modified<'2017-8-1') or
-> (t_modified >= '2018-7-1' and t_modified<'2018-8-1');
```

当然，如果你的系统上线时间更早，或者后面又插入了之后年份的数据的话，你就需要再把其他年份补齐。

到这里我给你说明了，由于加了`month()`函数操作，MySQL无法再使用索引快速定位功能，而只能使用全索引扫描。

不过优化器在这个问题上确实有“偷懒”行为，即使是对于不改变有序性的函数，也不会考虑使用索引。比如，对于`select * from tradelog where id + 1 = 10000`这个SQL语句，这个加1操作并不会改变有序性，但是MySQL优化器还是不能用`id`索引快速定位到9999这一行。所以，需要你在写SQL语句的时候，手动改写成`where id = 10000 - 1`才可以。

案例二：隐式类型转换

接下来我再跟你说一说，另一个经常让程序员掉坑里的例子。

我们一起看一下这条SQL语句：

```
mysql> select * from tradelog where tradeid=110717;
```

交易编号`tradeid`这个字段上，本来就有索引，但是`explain`的结果却显示，这条语句需要走全表扫描。你可能也发现了，`tradeid`的字段类型是`varchar(32)`，而输入的参数却是整型，所以需要做类型转换。

那么，现在这里就有两个问题：

1. 数据类型转换的规则是什么？
2. 为什么有数据类型转换，就需要走全索引扫描？

先来看第一个问题，你可能会说，数据库里面类型这么多，这种数据类型转换规则更多，我记不住，应该怎么办呢？

这里有一个简单的方法，看 `select "10" > 9` 的结果：

1. 如果规则是“将字符串转成数字”，那么就是做数字比较，结果应该是1；
2. 如果规则是“将数字转成字符串”，那么就是做字符串比较，结果应该是0。

验证结果如图3所示。

```
mysql> select "10" > 9;
+-----+
| "10" > 9 |
+-----+
|      1    |
+-----+
```

图3 MySQL中字符串和数字转换的效果示意图

从图中可知，`select "10" > 9` 返回的是1，所以你就能确认MySQL里的转换规则了：在MySQL中，字符串和数字做比较的话，是将字符串转换成数字。

这时，你再看这个全表扫描的语句：

```
mysql> select * from tradelog where tradeid=110717;
```

就知道对于优化器来说，这个语句相当于：

```
mysql> select * from tradelog where CAST(tradid AS signed int) = 110717;
```

也就是说，这条语句触发了我们上面说到的规则：对索引字段做函数操作，优化器会放弃走树搜索功能。

现在，我留给你一个小问题，`id`的类型是`int`，如果执行下面这个语句，是否会导致全表扫描呢？

```
select * from tradelog where id="83126";
```

你可以先自己分析一下，再到数据库里面去验证确认。

接下来，我们再来看一个稍微复杂点的例子。

案例三：隐式字符编码转换

假设系统里还有另外一个表trade_detail，用于记录交易的操作细节。为了便于量化分析和复现，我往交易日志表tradelog和交易详情表trade_detail这两个表里插入一些数据。

```
mysql> CREATE TABLE `trade_detail` (
    `id` int(11) NOT NULL,
    `tradeid` varchar(32) DEFAULT NULL,
    `trade_step` int(11) DEFAULT NULL, /*操作步骤*/
    `step_info` varchar(32) DEFAULT NULL, /*步骤信息*/
    PRIMARY KEY (`id`),
    KEY `tradeid` (`tradeid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
insert into tradelog values(1, 'aaaaaaaa', 1000, now());
insert into tradelog values(2, 'aaaaaaab', 1000, now());
insert into tradelog values(3, 'aaaaaaac', 1000, now());
```

```
insert into trade_detail values(1, 'aaaaaaaa', 1, 'add');
insert into trade_detail values(2, 'aaaaaaaa', 2, 'update');
insert into trade_detail values(3, 'aaaaaaaa', 3, 'commit');
insert into trade_detail values(4, 'aaaaaaab', 1, 'add');
insert into trade_detail values(5, 'aaaaaaab', 2, 'update');
insert into trade_detail values(6, 'aaaaaaab', 3, 'update again');
insert into trade_detail values(7, 'aaaaaaab', 4, 'commit');
insert into trade_detail values(8, 'aaaaaaac', 1, 'add');
insert into trade_detail values(9, 'aaaaaaac', 2, 'update');
insert into trade_detail values(10, 'aaaaaaac', 3, 'update again');
insert into trade_detail values(11, 'aaaaaaac', 4, 'commit');
```

这时候，如果要查询id=2的交易的所有操作步骤信息，SQL语句可以这么写：

```
mysql> select d.* from tradelog l, trade_detail d where d.tradeid=l.tradeid and l.id=2; /*语句Q1*/
```

mysql> explain select d.* from tradelog l , trade_detail d where d.tradeid=l.tradeid and l.id=2;												
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	l	NULL	const	PRIMARY,tradeid	PRIMARY	4	const	1	100.00	NULL	
1	SIMPLE	d	NULL	ALL	NULL	NULL	NULL	NULL	11	100.00	Using where	

图4 语句Q1的explain结果

我们一起来看下这个结果：

1. 第一行显示优化器会先在交易记录表tradelog上查到id=2的行，这个步骤用上了主键索引，rows=1表示只扫描一行；
2. 第二行key=NULL，表示没有用上交易详情表trade_detail上的tradeid索引，进行了全表扫描。

在这个执行计划里，是从tradelog表中取tradeid字段，再去trade_detail表里查询匹配字段。因此，我们把tradelog称为驱动表，把trade_detail称为被驱动表，把tradeid称为关联字段。

接下来，我们看下这个explain结果表示的执行流程：

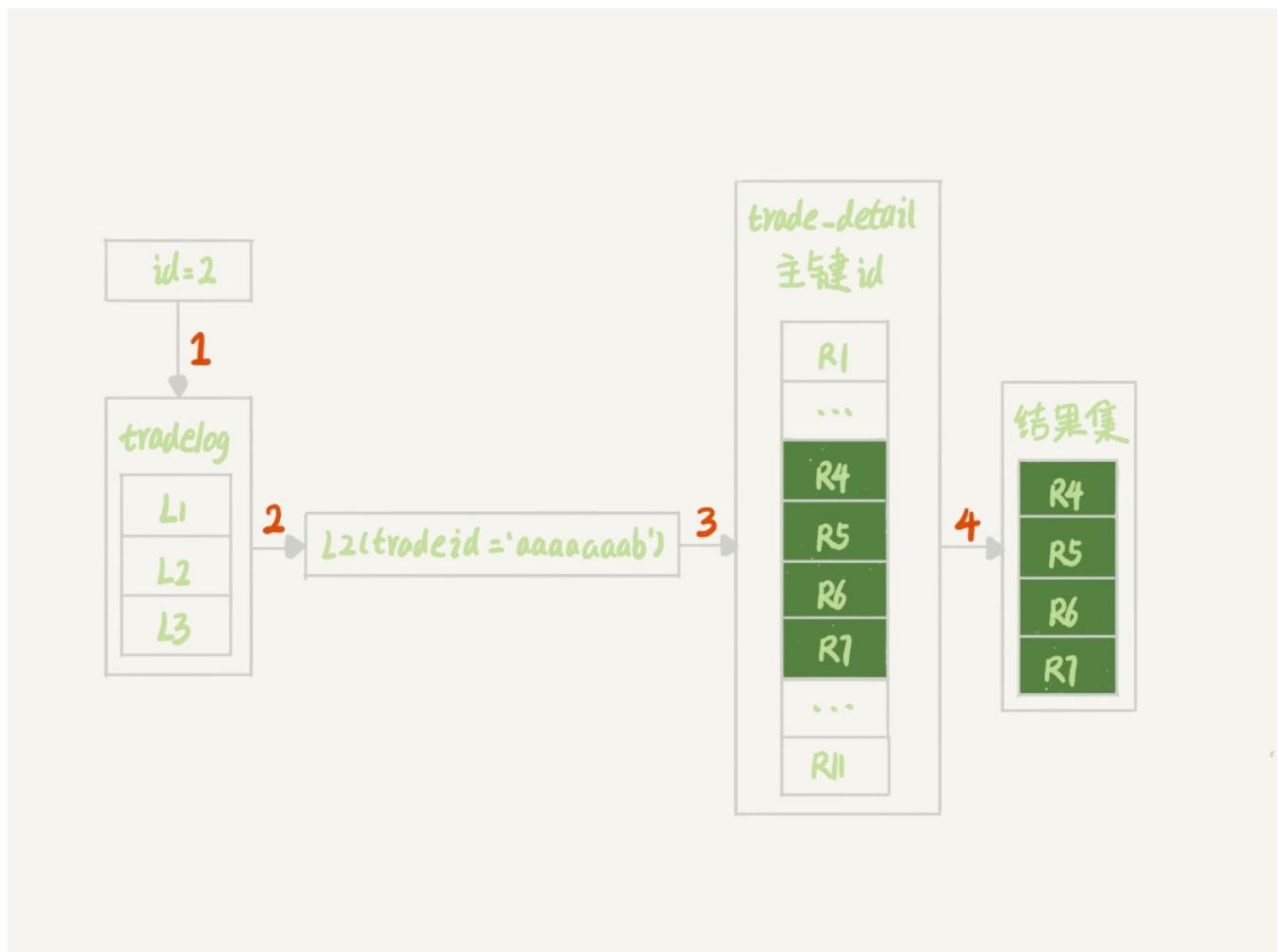


图5 语句Q1的执行过程

图中：

- 第1步，是根据id在tradelog表里找到L2这一行；
- 第2步，是从L2中取出tradeid字段的值；
- 第3步，是根据tradeid值到trade_detail表中查找条件匹配的行。`explain`的结果里面第二行的`key=NULL`表示的就是，这个过程是通过遍历主键索引的方式，一个一个地判断tradeid的值是否匹配。

进行到这里，你会发现第3步不符合我们的预期。因为表trade_detail里tradeid字段上是有索引的，我们本来是希望通过使用tradeid索引能够快速定位到等值的行。但，这里并没有。

如果你去问DBA同学，他们可能会告诉你，因为这两个表的字符集不同，一个是utf8，一个是utf8mb4，所以做表连接查询的时候用不上关联字段的索引。这个回答，也是通常你搜索这个问题时会得到的答案。

但是你应该再追问一下，为什么字符集不同就用不上索引呢？

我们说问题是出在执行步骤的第3步，如果单独把这一步改成SQL语句的话，那就是：

```
mysql> select * from trade_detail where tradeid=$L2.tradeid.value;
```

其中，\$L2.tradeid.value的字符集是utf8mb4。

参照前面的两个例子，你肯定就想到了，字符集utf8mb4是utf8的超集，所以当这两个类型的字符串在做比较的时候，MySQL内部的操作是，先把utf8字符串转成utf8mb4字符集，再做比较。

这个设定很好理解，utf8mb4是utf8的超集。类似地，在程序设计语言里面，做自动类型转换的时候，为了避免数据在转换过程中由于截断导致数据错误，也都是“按数据长度增加的方向”进行转换的。

因此，在执行上面这个语句的时候，需要将被驱动数据表里的字段一个个地转换成utf8mb4，再跟L2做比较。

也就是说，实际上这个语句等同于下面这个写法：

```
select * from trade_detail where CONVERT(tradeid USING utf8mb4)=$L2.tradeid.value;
```

CONVERT()函数，在这里的意思是把输入的字符串转成utf8mb4字符集。

这就再次触发了我们上面说到的原则：对索引字段做函数操作，优化器会放弃走树搜索功能。

到这里，你终于明确了，字符集不同只是条件之一，连接过程中要求在被驱动表的索引字段上加函数操作，是直接导致对被驱动表做全表扫描的原因。

作为对比验证，我给你提另外一个需求，“查找trade_detail表里id=4的操作，对应的操作者是谁”，再来看下这个语句和它的执行计划。

```
mysql>select l.operator from tradelog l , trade_detail d where d.tradeid=l.tradeid and d.id=4;
```

```
mysql> explain select l.operator from tradelog l , trade_detail d where d.tradeid=l.tradeid and d.id=4;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | d    | NULL       | const | PRIMARY      | PRIMARY | 4   | const | 1   | 100.00 | NULL  |
| 1 | SIMPLE     | l    | NULL       | ref   | tradeid      | tradeid | 131 | const | 1   | 100.00 | NULL  |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

图6 explain 结果

这个语句里trade_detail表成了驱动表，但是explain结果的第二行显示，这次的查询操作用上了被驱动表tradelog里的索引(tradeid)，扫描行数是1。

这也是两个tradeid字段的join操作，为什么这次能用上被驱动表的tradeid索引呢？我们来分析一下。

假设驱动表trade_detail里id=4的行记为R4，那么在连接的时候（图5的第3步），被驱动表tradelog上执行的就是类似这样的SQL语句：

```
select operator from tradelog where traideid =$R4.tradeid.value;
```

这时候\$R4.tradeid.value的字符集是utf8，按照字符集转换规则，要转成utf8mb4，所以这个过程就被改写成：

```
select operator from tradelog where traideid =CONVERT($R4.tradeid.value USING utf8mb4);
```

你看，这里的CONVERT函数是加在输入参数上的，这样就可以用上被驱动表的traideid索引。

理解了原理以后，就可以用来指导操作了。如果要优化语句

```
select d.* from tradelog l, trade_detail d where d.tradeid=l.tradeid and l.id=2;
```

的执行过程，有两种做法：

- 比较常见的优化方法是，把trade_detail表上的tradeid字段的字符集也改成utf8mb4，这样就没有字符集转换的问题了。

```
alter table trade_detail modify tradeid varchar(32) CHARACTER SET utf8mb4 default null;
```

- 如果能够修改字段的字符集的话，是最好不过了。但如果数据量比较大，或者业务上暂时不能做这个`DDL`的话，那就只能采用修改`SQL`语句的方法了。

```
mysql> select d.* from tradelog l , trade_detail d where d.tradeid=CONVERT(l.tradeid USING utf8) and l.id=2;
```

```
mysql> explain select d.* from tradelog l , trade_detail d where d.tradeid=CONVERT(l.tradeid USING utf8) and l.id=2;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | l      | NULL       | const  | PRIMARY        | PRIMARY | 4       | const | 1    | 100.00 | NULL   |
| 1 | SIMPLE     | d      | NULL       | ref    | tradeid        | tradeid | 99      | const | 4    | 100.00 | NULL   |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

图7 SQL语句优化后的explain结果

这里，我主动把`l.tradeid`转成`utf8`，就避免了被驱动表上的字符编码转换，从`explain`结果可以看到，这次索引走对了。

小结

今天我给你举了三个例子，其实是在说同一件事儿，即：对索引字段做函数操作，可能会破坏索引值的有序性，因此优化器就决定放弃走树搜索功能。

第二个例子是隐式类型转换，第三个例子是隐式字符编码转换，它们都跟第一个例子一样，因为要求在索引字段上做函数操作而导致了全索引扫描。

MySQL的优化器确实有“偷懒”的嫌疑，即使简单地把`where id+1=1000`改写成`where id=1000-1`就能够用上索引快速查找，也不会主动做这个语句重写。

因此，每次你的业务代码升级时，把可能出现的、新的`SQL`语句`explain`一下，是一个很好的习惯。

最后，又到了思考题时间。

今天我留给你的课后问题是，你遇到过别的、类似今天我们提到的性能问题吗？你认为原因是什么，又是怎么解决的呢？

你可以把你经历和分析写在留言区里，我会在下一篇文章的末尾选取有趣的评论跟大家一起分享和分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上篇文章的最后，留给你的问题是：我们文章中最后的一个方案是，通过三次`limit 1,1`来得

到需要的数据，你觉得有没有进一步的优化方法。

这里我给出一种方法，取Y1、Y2和Y3里面最大的一个数，记为M，最小的一个数记为N，然后执行下面这条SQL语句：

```
mysql> select * from t limit N, M-N+1;
```

再加上取整个表总行数的C行，这个方案的扫描行数总共只需要C+M+1行。

当然也可以先取回id值，在应用中确定了三个id值以后，再执行三次where id=X的语句也是可以的。@倪大人 同学在评论区就提到了这个方法。

这次评论区出现了很多很棒的留言：

@老杨同志 提出了重新整理的方法、@雪中鼠[悠闲] 提到了用rowid的方法，是类似的思路，就是让表里面保存一个无空洞的自增值，这样就可以用我们的随机算法1来实现；
@吴宇晨 提到了拿到第一个值以后，用id迭代往下找的方案，利用了主键索引的有序性。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Ding Qi, a man with glasses and a black shirt, standing with his arms crossed. To his left is the title "MySQL 实战 45 讲" and a subtitle "从原理到实战，丁奇带你搞懂 MySQL" above his photo. On the far left, there's a logo for "极客时间". Below the title, the author's name "林晓斌" is listed along with the note "网名丁奇 前阿里资深技术专家". At the bottom, there's a call-to-action button with the text "新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。".

精选留言



老杨同志

20

感谢老师鼓励，我本人工作时间比较长，有一定的基础，听老师的课还是收获很大。每次公司

内部有技术分享，我都去听课，但是多数情况，一两个小时的分享，就只有一两句话受益。老师的每篇文章都能命中我的知识盲点，感觉太别爽。

对应今天的隐式类型转换问题也踩过坑。

我们有个任务表记录待执行任务，表结构简化后如下：

```
CREATE TABLE `task` (
  `task_id` int(11) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `task_type` int(11) DEFAULT NULL COMMENT '任务类型id',
  `task_rfid` varchar(50) COLLATE utf8_unicode_ci DEFAULT NULL COMMENT '关联外键1',
  PRIMARY KEY (`task_id`)
) ENGINE=InnoDB AUTO_INCREMENT CHARSET=utf8 COLLATE=utf8_unicode_ci COMMENT='任务表';
```

task_rfid 是业务主键，当然都是数字，查询时使用sql：

```
select * from task where task_rfid =123;
```

其实这个语句也有隐式转换问题，但是待执行任务只有几千条记录，并没有什么感觉。

这个表还有个对应的历史表，数据有几千万

忽然有一天，想查一下历史记录，执行语句

```
select * from task_history where task_rfid =99;
```

直接就等待很长时间后超时报错了。

如果仔细看，其实我的表没有task_rfid 索引，写成task_rfid ='99'也一样是全表扫描。

运维时的套路是，猜测主键task_id的范围，怎么猜，我原表有creat_time字段，我会先查

```
select max(task_id) from task_history 然后再看看 select * from task_history where task_id = maxId - 10000的时间，估计出大概的id范围。然后语句变成
```

```
select * from task_history where task_rfid =99 and id between ? and ? ;
```

2018-12-24

| 作者回复

你最后这个id预估，加上between，

有种神来之笔的感觉！

感觉隐约里面有二分法的思想

||

2018-12-24



可凡不凡

1

1.老师好

2.如果在用一个 MySQL 关键字做字段,并且字段上索引,当我用这个索引作为唯一查询条件的时候,会造 成隐式的转换吗?

例如:SELECT * FROM b_side_order WHERE CODE = 332924 ; (code 上有索引)

3. mysql5.6 code 上有索引 intime 上没有索引

语句一:

```
SELECT * FROM b_side_order WHERE CODE = 332924 ;
```

语句二:

```
UPDATE b_side_order SET in_time = '2018-08-04 08:34:44' WHERE 1=2 or CODE = 332924;
```

这两个语句 执行计划走 `select` 走了索引,`update` 没有走索引 是执行计划的bug 吗??

2018-12-25

| 作者回复

1. 你好

2. `CODE`不是关键字呀， 另外优化器选择跟关键字无关哈，关键字的话，要用 反`括起来

3. 不是bug, `update`如果把 `or` 改成 `and` , 就能走索引

2018-12-25



冠超

0

非常感谢老师分享的内容，实打实地学到了。这里提个建议，希望老师能介绍一下设计表的时候要怎么考虑这方面的知识哈

2019-01-28

| 作者回复

是这样的，其实我们整个专栏大部分的文章，最后都是为了说明“怎么设计表”、“怎么考虑优化SQL语句”

但是因为这个不是一成不变的，很多是需要考虑现实的情况，

所以这个专栏就是想把对应的原理说一下，这样大家在应对不同场景的时候，可以组合来考虑

。

也就是说没有一段话可以把“怎么设计表”讲清楚（或者说硬写出来很可能就是一些general的没有什么针对性作用的描述）

你可以把你的业务背景抽象说下，我们来具体讨论吧

2019-01-28



700

0

老师您好，有个问题恳请指教。背景如下，我长话短说：

`mysql>select @@version;`

5.6.30-log

```
CREATE TABLE `t1` ( `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
`user_id` int(11) NOT NULL, `plan_id` int(11) NOT NULL DEFAULT '0' , PRIMARY KEY (`id`),
KEY `userid` (`user_id`) USING BTREE, KEY `idx_planid` (`plan_id`)
) ENGINE=InnoDB DEFAULT CHARSET=gb2312;
```

```
CREATE TABLE `t3` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `status` int(4) NOT NULL DEFAULT '0',
  `ootime` varchar(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_xxoo`(`status`, `ootime`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

t1 和 t3 表的字符集不一样

sql 执行计划如下：

```
explain
```

```
SELECT t1.id, t1.user_id
```

```
FROM t1, t3
```

```
WHERE t1.plan_id = t3.id
```

```
AND t3.ootime < UNIX_TIMESTAMP('2022-01-18')
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t3	index	PRIMARY	idx_xxoo	51	NULL	39106	Using where; Using index
1	SIMPLE	t1	ref	idx_planid	idx_planid	4	t3.id	401	Using join buffer (Batched Key Access)

我的疑惑是

1)t3 的 status 没出现在 where 条件中，但执行计划为什么用到了 idx_xxoo 紴引？

2)为什么 t3.ootime 也用到索引了，从 key_len 看出。t3.ootime 是 varchar 类型的，而 UNIX_TIMESTAMP('2022-01-18') 是数值，不是发生了隐式转换吗？

请老师指点。

2019-01-18

作者回复

这个查询语句会对t3做全索引扫描，是使用了索引的，只是没有用上快速搜索功能

2019-01-19



赖阿甘

0

"mysql>select l.operator from tradelog l , trade_detail d where d.tradeid=l.tradeid and d.id=4;"

图6上面那句sql是不是写错了。d.tradeid=l.tradeid是不是该写成l.tradeid = d.tradeid? 不然函数会作用在索引字段上，就只能全表扫描了

2018-12-24

作者回复

这个问题不是等号顺序决定的哈

好问题

2018-12-24



Leon

16

索引字段不能进行函数操作，但是索引字段的参数可以玩函数，一言以蔽之

2018-12-24

作者回复

精辟

2018-12-24



探索无止境

5

多表连接时，`mysql`是怎么选择驱动表和被驱动表的？这个很重要，希望老师可以讲讲

2018-12-25



可凡不凡

5

1. 老师对于多表联合查询中,MySQL 对索引的选择 以后会详细介绍吗?

2018-12-24

作者回复

额，你是第三个提这个问题的了，我得好好考虑下安排

2018-12-24



某、人

4

SQL逻辑相同,性能差异较大的,通过老师所讲学习到的,和平时碰到的,大概有以下几类:

一. 字段发生了转换,导致本该使用索引而没有用到索引

1. 条件字段函数操作

2. 隐式类型转换

3. 隐式字符编码转换

(如果驱动表的字符集比被驱动表得字符集小，关联列就能用到索引,如果更大,需要发生隐式编码转换,则不能用到索引,`latin<gbk<utf8<utf8mb4`)

二. 嵌套循环,驱动表与被驱动表选择错误

1. 连接列上没有索引,导致大表驱动小表,或者小表驱动大表(但是大表走的是全表扫描) --连接列上建立索引

2. 连接列上虽然有索引,但是驱动表任然选择错误。 --通过`straight_join`强制选择关联表顺序

3. 子查询导致先执行外表在执行子查询,也是驱动表与被驱动表选择错误。

--可以考虑把子查询改写为内连接,或者改写内联视图(子查询放在`from`后组成一个临时表,在于其他表进行关联)

4.只需要内连接的语句,但是写成了左连接或者右连接。比如`select * from t left join b on t.id=b.id where b.name='abc'`驱动表被固定,大概率会扫描更多的行,导致效率降低。

--根据业务情况或sql情况,把左连接或者右连接改写为内连接

三.索引选择不同,造成性能差异较大

1.`select * from t where aid= and create_name>" order by id limit 1;`

选择走id索引或者选择走(aid,create_time)索引,性能差异较大.结果集都有可能不一致

--这个可以通过where条件过滤的值多少来大概判断,该走哪个索引

四.其它一些因素

1.比如之前学习到的是是否有MDL X锁

2.`innodb_buffer_pool`设置得太小,`innodb_io_capacity`设置得太小,刷脏速度跟不上

3.是否是对表做了DML语句之后,马上做`select`,导致`change buffer`收益不高

4.是否有数据空洞

5.`select`选取的数据是否在`buffer_pool`中

6.硬件原因,资源抢占

原因多种多样,还需要慢慢补充。

老师我问一个问题:

连接列上一个是int一个是bigint或者一个是char一个varchar,为什么被驱动表上会出现(using index condition)?

2018-12-24



Destroy、

2

老师,对于最后回答上一课的问题: `mysql> select * from t limit N, M-N+1;`
这个语句也不是取3条记录。没理解。

2018-12-27

| 作者回复

取其中三条...

2018-12-27



风轨

2

刚试了文中穿插得思考题:当主键是整数类型条件是字符串时,会走索引。

文中提到了当字符串和数字比较时会把字符串转化为数字,所以隐式转换不会应用到字段上,所以可以走索引。

另外, `select 'a' = 0 ;` 的结果是1,说明无法转换成数字的字符串都被转换成0来处理了。

2018-12-24

| 作者回复

||

2018-12-24



匿名的朋友

1

丁奇老师,我有个疑问,就是sql语句执行时那些`order by group by limit`以及`where`条件,有执行的先后顺序吗?

2019-01-05

作者回复

有，先**where** ,再**order by** 最后**limit**

2019-01-05



大坤

1

之前遇到过按时间范围查询大表不走索引的情况，如果缩小时间范围，又会走索引，记得在一些文章中看到过结果数据超过全表的30%就会走全表扫描，但是前面说的时间范围查询大表，这个时间范围绝对是小于30%的情况，想请教下老师，这个优化器都是在什么情况下会放弃索引呢？

2018-12-25

作者回复

总体来说就是判断哪种方式消耗更小，选哪种

2018-12-25



Leon

1

老师，经常面试被问到工作中做了什么优化，有没有好的业务表的设计，请问老师课程结束后能不能给我们一个提纲挈领的大纲套路，让我们有个脉络和思路来应付这种面试套路

2018-12-25

作者回复

有没有好的业务表的设计，这类问题我第一次听到，能不能展开一下，这样说不要清楚面试官的考核点是啥...

2018-12-25



果然如此

1

我想问一个上期的问题，随机算法2虽然效率高，但是还是有个瑕疵，比如我们的随机出题算法无法直接应用，因为每次随机一个试题id，多次随机没有关联，会产生重复id，有没有更好的解决方法？

2018-12-25

作者回复

内存里准备个**set**这样的数据结构，重读的不算，这样可以不

2018-12-25



长杰

1

这里我给出一种方法，取 Y1、Y2 和 Y3 里面最大的一个数，记为 M，最小的一个数记为 N，然后执行下面这条 SQL 语句：

```
mysql> select * from t limit N, M-N+1;
```

再加上取整个表总行数的 C 行，这个方案的扫描行数总共只需要 C+M 行。

优化后的方案应该是 C+M+1 行吧？

2018-12-24

作者回复

你说的对，我改下

2018-12-25



asdf100

1

在这个例子里，放弃了树搜索功能，优化器可以选择遍历主键索引，也可以选择遍历索引 `t_modified`，优化器对比索引大小后发现，索引 `t_modified` 更小，遍历这个索引比遍历主键索引来得更快。

优化器如何对比的，根据参与字段字段类型占用空间大小吗？

2018-12-24

| 作者回复

优化器信息是引擎给的，

引擎是这么判断的

2018-12-24



约书亚

1

谁是驱动表谁是被驱动表，是否大多数情况看 `where` 条件就可以了？这是否本质上涉及到 MySQL 底层决定用什么算法进行级联查询的问题？后面会有课程详细说明嘛？

2018-12-24

| 作者回复

可以简单看 `where` 之后剩下的行数（预判不一定准哈）

2018-12-24



Lukia

0

老师好，之前看了《数据索引与优化》，提到表之间的连接操作可以有嵌套循环连接（本文中提到的驱动表和被驱动表）和合并扫描连接（先在临时表中针对谓词作排序）还有哈希连接。请问 MySQL 中是否存在后面两种方式的连接，如果有的话优化器会在什么情况下选择呢？谢谢！

2019-01-29

| 作者回复

第34、35两篇就会说到了，今晚关注下

2019-01-29



涛哥哥

0

老师，您好！我是做后端开发的。想问一下 mysql `in` 关键字 的内部原理，能抽一点点篇幅讲一下吗？比如：`select * from T where id in (a,b,d,c,,e,f);` `id` 是主键。1、为什么查询出来的结果集会按照 `id` 排一次序呢（是跟去重有关系么）？2、如果 `in` 里面的值较多的时候，就会比较慢啊（是还不如全表扫描么）？问我们公司很多后端的，都不太清楚，问我们 DBA，他说默认就是这样（这不跟没说一样吗）。希望老师可以帮忙解惑。祝老师身体健康！微笑~

2019-01-26

| 作者回复

1. 优化器会排个序，目的是如果这几个记录对应的数据都不在内存里，可以触发顺序读盘，后面文章我们介绍到 `join` 的时候，会提到 MRR，你关注下

2. in里面值多就是多次执行树搜索，跟全表扫描的速度对比，就看in里面的数据个数的比例了。
你的in里面一般多少个value呀

2019-01-26

19 | 为什么我只查一行的语句，也执行这么慢？

2018-12-26 林晓斌



一般情况下，如果我跟你说查询性能优化，你首先会想到一些复杂的语句，想到查询需要返回大量的数据。但有些情况下，“查一行”，也会执行得特别慢。今天，我就跟你聊聊这个有趣的话题，看看什么情况下，会出现这个现象。

需要说明的是，如果MySQL数据库本身就有很大的压力，导致数据库服务器CPU占用率很高或ioutil (IO利用率) 很高，这种情况下所有语句的执行都有可能变慢，不属于我们今天的讨论范围。

为了便于描述，我还是构造一个表，基于这个表来说明今天的问题。这个表有两个字段id和c，并且我在里面插入了10万行记录。

```
mysql> CREATE TABLE `t` (
    `id` int(11) NOT NULL,
    `c` int(11) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;

delimiter ;;
create procedure idata()
begin
    declare i int;
    set i=1;
    while(i<=100000)do
        insert into t values(i,i)
        set i=i+1;
    end while;
end;;
delimiter ;

call idata();
```

接下来，我会用几个不同的场景来举例，有些是前面的文章中我们已经介绍过的知识点，你看看能不能一眼看穿，来检验一下吧。

第一类：查询长时间不返回

如图1所示，在表t执行下面的SQL语句：

```
mysql> select * from t where id=1;
```

查询结果长时间不返回。

```
mysql> select * from t where id=1;
```

图1 查询长时间不返回

一般碰到这种情况的话，大概率是表t被锁住了。接下来分析原因的时候，一般都是首先执行一下**show processlist**命令，看看当前语句处于什么状态。

然后我们再针对每种状态，去分析它们产生的原因、如何复现，以及如何处理。

等MDL锁

如图2所示，就是使用show processlist命令查看Waiting for table metadata lock的示意图。

mysql> show processlist;						
ID	User	Host	db	Command	Time	State
5	root	localhost:61558	test	Query	0	init
7	root	localhost:63852	test	Sleep	31	
8	root	localhost:63870	test	Query	25	Waiting for table metadata lock

图2 Waiting for table metadata lock状态示意图

出现这个状态表示的是，现在有一个线程正在表t上请求或者持有MDL写锁，把select语句堵住了。

在第6篇文章 [《全局锁和表锁：给表加个字段怎么有这么多阻碍？》](#) 中，我给你介绍过一种复现方法。但需要说明的是，那个复现过程是基于MySQL 5.6版本的。而MySQL 5.7版本修改了MDL的加锁策略，所以就不能复现这个场景了。

不过，在MySQL 5.7版本下复现这个场景，也很容易。如图3所示，我给出了简单的复现步骤。

session A	session B
lock table t write; 	 select * from t where id=1;

图3 MySQL 5.7中Waiting for table metadata lock的复现步骤

session A 通过lock table命令持有表t的MDL写锁，而session B的查询需要获取MDL读锁。所以，session B进入等待状态。

这类问题的处理方式，就是找到谁持有MDL写锁，然后把它kill掉。

但是，由于在show processlist的结果里面，session A的Command列是“Sleep”，导致查找起来很不方便。不过有了performance_schema和sys系统库以后，就方便多了。（MySQL启动时需要设置performance_schema=on）

通过查询sys.schema_table_lock_waits这张表，我们就可以直接找出造成阻塞的process id，把这个连接用kill命令断开即可。

```
mysql> select blocking_pid from sys.schema_table_lock_waits;
+-----+
| blocking_pid |
+-----+
|      4      |
+-----+
```

图4 查获加表锁的线程id

等flush

接下来，我给你举另外一种查询被堵住的情况。

我在表t上，执行下面的SQL语句：

```
mysql> select * from information_schema.processlist where id=1;
```

这里，我先卖个关子。

你可以看一下图5。我查出来这个线程的状态是Waiting for table flush，你可以设想一下这是什么原因。

```
mysql> select * from information_schema.processlist where id=6;
+----+----+----+----+----+----+----+----+
| ID | USER | HOST          | DB   | COMMAND | TIME | STATE        | INFO
+----+----+----+----+----+----+----+----+
|  6 | root  | localhost:47074 | test | Query   |  622 | Waiting for table flush | select * from t where id=1 |
+----+----+----+----+----+----+----+----+
```

图5 Waiting for table flush状态示意图

这个状态表示的是，现在有一个线程正要对表t做flush操作。MySQL里面对表做flush操作的用法，一般有以下两个：

```
flush tables t with read lock;
```

```
flush tables with read lock;
```

这两个flush语句，如果指定表t的话，代表的是只关闭表t；如果没有指定具体的表名，则表示关闭MySQL里所有打开的表。

但是正常这两个语句执行起来都很快，除非它们也被别的线程堵住了。

所以，出现Waiting for table flush状态的可能情况是：有一个flush tables命令被别的语句堵住

了，然后它又堵住了我们的**select**语句。

现在，我们一起来复现一下这种情况，复现步骤如图6所示：

session A	session B	session C
select sleep(1) from t;		
	flush tables t;	
		select * from t where id=1;

图6 Waiting for table flush的复现步骤

在**session A**中，我故意每行都调用一次**sleep(1)**，这样这个语句默认要执行10万秒，在这期间表t一直是被**session A**“打开”着。然后，**session B**的**flush tables t**命令再要去关闭表t，就需要等**session A**的查询结束。这样，**session C**要再次查询的话，就会被**flush**命令堵住了。

图7是这个复现步骤的**show processlist**结果。这个例子的排查也很简单，你看到这个**show processlist**的结果，肯定就知道应该怎么做了。

mysql> show processlist;						
Id	User	Host	db	Command	Time	State
4	root	localhost:49548	test	Query	38	User sleep
5	root	localhost:49604	test	Query	35	Waiting for table flush
6	root	localhost:49634	test	Query	30	Waiting for table flush
7	root	localhost:49726	test	Query	0	starting

图 7 Waiting for table flush的show processlist 结果

等行锁

现在，经过了表级锁的考验，我们的**select**语句终于来到引擎里了。

```
mysql> select * from t where id=1 lock in share mode;
```

上面这条语句的用法你也很熟悉了，我们在第8篇[《事务到底是隔离的还是不隔离的？》](#)文章介绍当前读时提到过。

由于访问**id=1**这个记录时要加读锁，如果这时候已经有一个事务在这行记录上持有一个写锁，我们的**select**语句就会被堵住。

复现步骤和现场如下：

session A	session B
begin; update t set c=c+1 where id=1;	
	select * from t where id=1 lock in share mode;

图 8 行锁复现

```
mysql> show processlist;
+----+----+-----+----+----+----+----+----+
| Id | User | Host      | db | Command | Time | State    | Info
+----+----+-----+----+----+----+----+----+
|  4 | root | localhost:65224 | test | Query   |    0 | starting | show processlist
|  8 | root | localhost:10354 | test | Query   |    1 | statistics | select * from t where id=1 lock in share mode
| 10 | root | localhost:11276 | test | Sleep   |   52 |          | NULL
+----+----+-----+----+----+----+----+----+
3 rows in set (0.00 sec)
```

图 9 行锁show processlist 现场

显然，**session A**启动了事务，占有写锁，还不提交，是导致**session B**被堵住的原因。

这个问题并不难分析，但问题是怎麽查出是谁占着这个写锁。如果你用的是MySQL 5.7版本，可以通过**sys.innodb_lock_waits**表查到。

查询方法是：

```
mysql> select * from sys.innodb_lock_waits where locked_table='test'.'t'\G
```

```
mysql> select * from sys.innodb_lock_waits where locked_table='`test`.`t`\G
***** 1. row *****
    wait_started: 2018-12-13 20:12:35
        wait_age: 00:00:08
    wait_age_secs: 8
        locked_table: `test`.`t`
        locked_index: PRIMARY
        locked_type: RECORD
    waiting_trx_id: 421668144410224
    waiting trx started: 2018-12-13 20:12:35
        waiting trx age: 00:00:08
    waiting trx rows_locked: 1
    waiting trx rows_modified: 0
        waiting pid: 8
            waiting_query: select * from t where id=1 lock in share mode
            waiting_lock_id: 421668144410224:23:4:2
            waiting_lock_mode: S
            blocking trx id: 1101302
                blocking pid: 4
                blocking_query: NULL
            blocking_lock_id: 1101302:23:4:2
            blocking_lock_mode: X
            blocking trx started: 2018-12-13 20:01:57
                blocking trx age: 00:10:46
            blocking trx rows_locked: 1
            blocking trx rows_modified: 1
            sql_kill_blocking_query: KILL QUERY 4
sql_kill_blocking_connection: KILL 4
1 row in set, 3 warnings (0.00 sec)
```

图10 通过sys.innodb_lock_waits 查行锁

可以看到，这个信息很全，4号线程是造成堵塞的罪魁祸首。而干掉这个罪魁祸首的方式，就是KILL QUERY 4或KILL 4。

不过，这里不应该显示“KILL QUERY 4”。这个命令表示停止4号线程当前正在执行的语句，而这个方法其实是没有用的。因为占有行锁的是update语句，这个语句已经是之前执行完成了的，现在执行KILL QUERY，无法让这个事务去掉id=1上的行锁。

实际上，KILL 4才有效，也就是说直接断开这个连接。这里隐含的一个逻辑就是，连接被断开的时候，会自动回滚这个连接里面正在执行的线程，也就释放了id=1上的行锁。

第二类：查询慢

经过了重重封“锁”，我们再来看看一些查询慢的例子。

先来看一条你一定知道原因的SQL语句：

```
mysql> select * from t where c=50000 limit 1;
```

由于字段c上没有索引，这个语句只能走id主键顺序扫描，因此需要扫描5万行。

作为确认，你可以看一下慢查询日志。注意，这里为了把所有语句记录到slow log里，我在连接后先执行了 set long_query_time=0，将慢查询日志的时间阈值设置为0。

```
# Query_time: 0.011543 Lock_time: 0.000104 Rows_sent: 1 Rows_examined: 50000
SET timestamp=1544723147;
select * from t where c=50000 limit 1;
```

图11 全表扫描5万行的slow log

Rows_examined显示扫描了50000行。你可能会说，不是很慢呀，11.5毫秒就返回了，我们线上一般都配置超过1秒才算慢查询。但你要记住：坏查询不一定是慢查询。我们这个例子里面只有10万行记录，数据量大起来的话，执行时间就线性涨上去了。

扫描行数多，所以执行慢，这个很好理解。

但是接下来，我们再看一个只扫描一行，但是执行很慢的语句。

如图12所示，是这个例子的slow log。可以看到，执行的语句是

```
mysql> select * from t where id=1;
```

虽然扫描行数是1，但执行时间却长达800毫秒。

```
# User@Host: root[root] @ localhost [127.0.0.1] Id:      5
# Query_time: 0.804400 Lock_time: 0.000205 Rows_sent: 1 Rows_examined: 1
SET timestamp=1544728393;
```

图12 扫描一行却执行得很慢

是不是有点奇怪呢，这些时间都花在哪里了？

如果我把这个slow log的截图再往下拉一点，你可以看到下一个语句，select * from t where id=1 lock in share mode，执行时扫描行数也是1行，执行时间是0.2毫秒。

```
# Query_time: 0.000258 Lock_time: 0.000132 Rows_sent: 1 Rows_examined: 1
SET timestamp=1544728398;
select * from t where id=1 lock in share mode;
```

图 13 加上lock in share mode的slow log

看上去是不是更奇怪了？按理说lock in share mode还要加锁，时间应该更长才对啊。

可能有的同学已经有答案了。如果你还没有答案的话，我再给你一个提示信息，图14是这两个

语句的执行输出结果。

```
mysql> select * from t where id=1;
+----+---+
| id | c   |
+----+---+
| 1  | 1   |
+----+---+
1 row in set (0.81 sec)

mysql> select * from t where id=1 lock in share mode;
+----+-----+
| id | c       |
+----+-----+
| 1  | 1000001 |
+----+-----+
1 row in set (0.00 sec)
```

图14 两个语句的输出结果

第一个语句的查询结果里c=1，带lock in share mode的语句返回的是c=1000001。看到这里应该有更多的同学知道原因了。如果你还是没有头绪的话，也别着急。我先跟你说明一下复现步骤，再分析原因。

session A	session B
start transaction with consistent snapshot;	
	update t set c=c+1 where id=1; //执行100万次
select * from t where id=1;	
select * from t where id=1 lock in share mode;	

图15 复现步骤

你看到了，session A先用start transaction with consistent snapshot命令启动了一个事务，之后session B才开始执行update语句。

session B执行完100万次update语句后，id=1这一行处于什么状态呢？你可以从图16中找到答案。

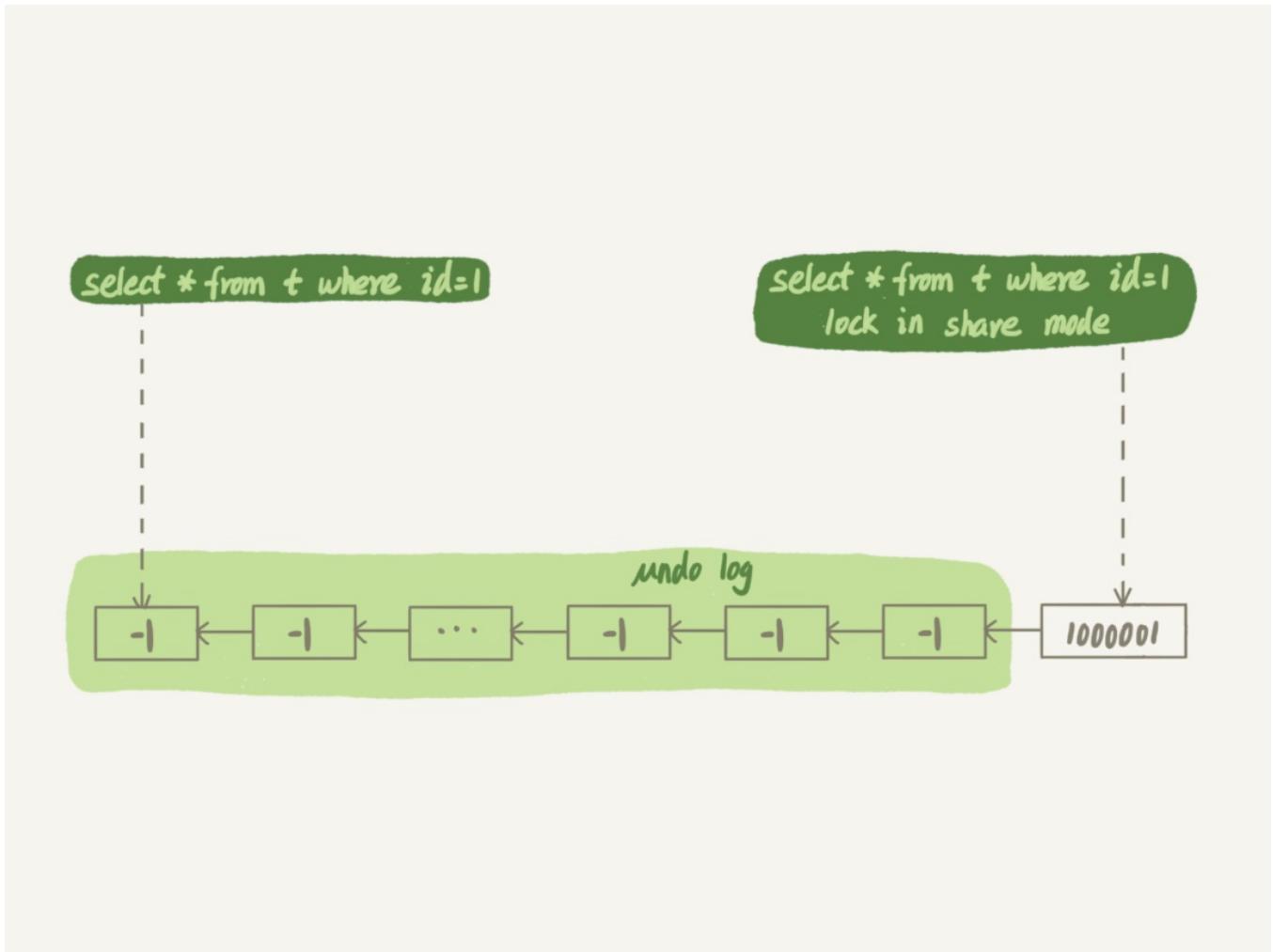


图16 `id=1`的数据状态

session B更新完100万次，生成了100万个回滚日志(undo log)。

带`lock in share mode`的SQL语句，是当前读，因此会直接读到`1000001`这个结果，所以速度很快；而`select * from t where id=1`这个语句，是一致性读，因此需要从`1000001`开始，依次执行`undo log`，执行了100万次以后，才将`1`这个结果返回。

注意，`undo log`里记录的其实是“把2改成1”，“把3改成2”这样的操作逻辑，画成减1的目的是方便你看图。

小结

今天我给你举了一个简单的表上，执行“查一行”，可能出现的被锁住和执行慢的例子。这其中涉及到了表锁、行锁和一致性读的概念。

在实际使用中，碰到的场景会更复杂。但大同小异，你可以按照我在文章中介绍的定位方法，来定位并解决问题。

最后，我给你留一个问题吧。

我们在举例加锁读的时候，用的是这个语句，`select * from t where id=1 lock in share mode`。由

于**id**上有索引，所以可以直接定位到**id=1**这一行，因此读锁也是只加在了这一行上。

但如果是下面的**SQL**语句，

```
begin;
select * from t where c=5 for update;
commit;
```

这个语句序列是怎么加锁的呢？加的锁又是什么时候释放呢？

你可以把你的观点和验证方法写在留言区里，我会在下一篇文章的末尾给出我的参考答案。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期间题时间

在上一篇文章最后，我留给你的问题是，希望你可以分享一下之前碰到过的、与文章中类似的场景。

@封建的风 提到一个有趣的场景，值得一说。我把他的问题重写一下，表结构如下：

```
mysql> CREATE TABLE `table_a` (
    `id` int(11) NOT NULL,
    `b` varchar(10) DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY `b` (`b`)
) ENGINE=InnoDB;
```

假设现在表里面，有100万行数据，其中有10万行数据的**b**的值是'1234567890'，假设现在执行语句是这么写的：

```
mysql> select * from table_a where b='1234567890abcd';
```

这时候，**MySQL**会怎么执行呢？

最理想的情况是，**MySQL**看到字段**b**定义的是**varchar(10)**，那肯定返回空呀。可惜，**MySQL**并没有这么做。

那要不，就是把'1234567890abcd'拿到索引里面去做匹配，肯定也没能够快速判断出索引树**b**上并没有这个值，也很快就能返回空结果。

但实际上，MySQL也不是这么做的。

这条SQL语句的执行很慢，流程是这样的：

1. 在传给引擎执行的时候，做了字符截断。因为引擎里面这个行只定义了长度是10，所以只截了前10个字节，就是'1234567890'进去做匹配；
2. 这样满足条件的数据有10万行；
3. 因为是select *， 所以要做10万次回表；
4. 但是每次回表以后查出整行，到server层一判断，b的值都不是'1234567890abcd'；
5. 返回结果是空。

这个例子，是我们文章内容的一个很好的补充。虽然执行过程中可能经过函数操作，但是最终在拿到结果后，server层还是要做一轮判断的。

评论区留言点赞板：

@赖阿甘 提到了等号顺序问题，时间上MySQL优化器执行过程中，where 条件部分，`a=b`和`b=a`的写法是一样的。

@沙漠里的骆驼 提到了一个常见的问题。相同的模板语句，但是匹配行数不同，语句执行时间相差很大。这种情况，在语句里面有`order by`这样的操作时会更明显。

@Justin 回答了我们正文中的问题，如果`id` 的类型是整数，传入的参数类型是字符串的时候，可以用上索引。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。

19 | 为什么我只查一行的语句，也执行这么慢？

2018-12-26 林晓斌



一般情况下，如果我跟你说查询性能优化，你首先会想到一些复杂的语句，想到查询需要返回大量的数据。但有些情况下，“查一行”，也会执行得特别慢。今天，我就跟你聊聊这个有趣的话题，看看什么情况下，会出现这个现象。

需要说明的是，如果MySQL数据库本身就有很大的压力，导致数据库服务器CPU占用率很高或ioutil (IO利用率) 很高，这种情况下所有语句的执行都有可能变慢，不属于我们今天的讨论范围。

为了便于描述，我还是构造一个表，基于这个表来说明今天的问题。这个表有两个字段id和c，并且我在里面插入了10万行记录。

```
mysql> CREATE TABLE `t` (
    `id` int(11) NOT NULL,
    `c` int(11) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

```
delimiter ;;
create procedure idata()
begin
declare i int;
set i=1;
while(i<=100000)do
    insert into t values(i,i);
    set i=i+1;
end while;
end;;
delimiter ;

call idata();
```

接下来，我会用几个不同的场景来举例，有些是前面的文章中我们已经介绍过的知识点，你看看能不能一眼看穿，来检验一下吧。

第一类：查询长时间不返回

如图1所示，在表t执行下面的SQL语句：

```
mysql> select * from t where id=1;
```

查询结果长时间不返回。

```
mysql> select * from t where id=1;
```

图1 查询长时间不返回

一般碰到这种情况的话，大概率是表t被锁住了。接下来分析原因的时候，一般都是首先执行一下**show processlist**命令，看看当前语句处于什么状态。

然后我们再针对每种状态，去分析它们产生的原因、如何复现，以及如何处理。

等MDL锁

如图2所示，就是使用show processlist命令查看Waiting for table metadata lock的示意图。

mysql> show processlist;						
ID	User	Host	db	Command	Time	State
5	root	localhost:61558	test	Query	0	init
7	root	localhost:63852	test	Sleep	31	
8	root	localhost:63870	test	Query	25	Waiting for table metadata lock

图2 Waiting for table metadata lock状态示意图

出现这个状态表示的是，现在有一个线程正在表t上请求或者持有MDL写锁，把select语句堵住了。

在第6篇文章 [《全局锁和表锁：给表加个字段怎么有这么多阻碍？》](#) 中，我给你介绍过一种复现方法。但需要说明的是，那个复现过程是基于MySQL 5.6版本的。而MySQL 5.7版本修改了MDL的加锁策略，所以就不能复现这个场景了。

不过，在MySQL 5.7版本下复现这个场景，也很容易。如图3所示，我给出了简单的复现步骤。

session A	session B
lock table t write; 	 select * from t where id=1;

图3 MySQL 5.7中Waiting for table metadata lock的复现步骤

session A 通过lock table命令持有表t的MDL写锁，而session B的查询需要获取MDL读锁。所以，session B进入等待状态。

这类问题的处理方式，就是找到谁持有MDL写锁，然后把它kill掉。

但是，由于在show processlist的结果里面，session A的Command列是“Sleep”，导致查找起来很不方便。不过有了performance_schema和sys系统库以后，就方便多了。（MySQL启动时需要设置performance_schema=on，相比于设置为off会有10%左右的性能损失）

通过查询sys.schema_table_lock_waits这张表，我们就可以直接找出造成阻塞的process id，把这个连接用kill命令断开即可。

```
mysql> select blocking_pid from sys.schema_table_lock_waits;
+-----+
| blocking_pid |
+-----+
|        4   |
+-----+
```

图4 查获加表锁的线程id

等flush

接下来，我给你举另外一种查询被堵住的情况。

我在表t上，执行下面的SQL语句：

```
mysql> select * from information_schema.processlist where id=1;
```

这里，我先卖个关子。

你可以看一下图5。我查出来这个线程的状态是Waiting for table flush，你可以设想一下这是什么原因。

```
mysql> select * from information_schema.processlist where id=6;
+----+----+----+----+----+----+----+----+
| ID | USER | HOST          | DB    | COMMAND | TIME | STATE           | INFO
+----+----+----+----+----+----+----+----+
|  6 | root  | localhost:47074 | test  | Query   |  622 | Waiting for table flush | select * from t where id=1 |
+----+----+----+----+----+----+----+----+
```

图5 Waiting for table flush状态示意图

这个状态表示的是，现在有一个线程正要对表t做flush操作。MySQL里面对表做flush操作的用法，一般有以下两个：

```
flush tables t with read lock;
```

```
flush tables with read lock;
```

这两个flush语句，如果指定表t的话，代表的是只关闭表t；如果没有指定具体的表名，则表示关闭MySQL里所有打开的表。

但是正常这两个语句执行起来都很快，除非它们也被别的线程堵住了。

所以，出现Waiting for table flush状态的可能情况是：有一个flush tables命令被别的语句堵住

了，然后它又堵住了我们的**select**语句。

现在，我们一起来复现一下这种情况，复现步骤如图6所示：

session A	session B	session C
select sleep(1) from t;		
	flush tables t;	
		select * from t where id=1;

图6 Waiting for table flush的复现步骤

在**session A**中，我故意每行都调用一次**sleep(1)**，这样这个语句默认要执行10万秒，在这期间表t一直是被**session A**“打开”着。然后，**session B**的**flush tables t**命令再要去关闭表t，就需要等**session A**的查询结束。这样，**session C**要再次查询的话，就会被**flush**命令堵住了。

图7是这个复现步骤的**show processlist**结果。这个例子的排查也很简单，你看到这个**show processlist**的结果，肯定就知道应该怎么做了。

mysql> show processlist;						
Id	User	Host	db	Command	Time	State
4	root	localhost:49548	test	Query	38	User sleep
5	root	localhost:49604	test	Query	35	Waiting for table flush
6	root	localhost:49634	test	Query	30	Waiting for table flush
7	root	localhost:49726	test	Query	0	starting

图 7 Waiting for table flush的show processlist 结果

等行锁

现在，经过了表级锁的考验，我们的**select**语句终于来到引擎里了。

```
mysql> select * from t where id=1 lock in share mode;
```

上面这条语句的用法你也很熟悉了，我们在第8篇[《事务到底是隔离的还是不隔离的？》](#)文章介绍当前读时提到过。

由于访问**id=1**这个记录时要加读锁，如果这时候已经有一个事务在这行记录上持有一个写锁，我们的**select**语句就会被堵住。

复现步骤和现场如下：

session A	session B
begin; update t set c=c+1 where id=1;	
	select * from t where id=1 lock in share mode;

图 8 行锁复现

```
mysql> show processlist;
+----+----+-----+----+----+----+----+----+
| Id | User | Host      | db | Command | Time | State    | Info
+----+----+-----+----+----+----+----+----+
|  4 | root | localhost:65224 | test | Query   |    0 | starting | show processlist
|  8 | root | localhost:10354 | test | Query   |    1 | statistics | select * from t where id=1 lock in share mode
| 10 | root | localhost:11276 | test | Sleep   |   52 |          | NULL
+----+----+-----+----+----+----+----+----+
3 rows in set (0.00 sec)
```

图 9 行锁show processlist 现场

显然，**session A**启动了事务，占有写锁，还不提交，是导致**session B**被堵住的原因。

这个问题并不难分析，但问题是怎麽查出是谁占着这个写锁。如果你用的是MySQL 5.7版本，可以通过**sys.innodb_lock_waits**表查到。

查询方法是：

```
mysql> select * from sys.innodb_lock_waits where locked_table='test.t'\G
```

```
mysql> select * from sys.innodb_lock_waits where locked_table='`test`.`t`\G
***** 1. row *****
    wait_started: 2018-12-13 20:12:35
        wait_age: 00:00:08
    wait_age_secs: 8
        locked_table: `test`.`t`
        locked_index: PRIMARY
        locked_type: RECORD
    waiting_trx_id: 421668144410224
    waiting trx started: 2018-12-13 20:12:35
        waiting trx age: 00:00:08
    waiting trx rows_locked: 1
    waiting trx rows_modified: 0
        waiting pid: 8
            waiting_query: select * from t where id=1 lock in share mode
            waiting_lock_id: 421668144410224:23:4:2
            waiting_lock_mode: S
            blocking trx id: 1101302
                blocking pid: 4
                blocking_query: NULL
            blocking_lock_id: 1101302:23:4:2
            blocking_lock_mode: X
            blocking trx started: 2018-12-13 20:01:57
                blocking trx age: 00:10:46
            blocking trx rows_locked: 1
            blocking trx rows_modified: 1
            sql_kill_blocking_query: KILL QUERY 4
sql_kill_blocking_connection: KILL 4
1 row in set, 3 warnings (0.00 sec)
```

图10 通过sys.innodb_lock_waits 查行锁

可以看到，这个信息很全，4号线程是造成堵塞的罪魁祸首。而干掉这个罪魁祸首的方式，就是KILL QUERY 4或KILL 4。

不过，这里不应该显示“KILL QUERY 4”。这个命令表示停止4号线程当前正在执行的语句，而这个方法其实是没有用的。因为占有行锁的是update语句，这个语句已经是之前执行完成了的，现在执行KILL QUERY，无法让这个事务去掉id=1上的行锁。

实际上，KILL 4才有效，也就是说直接断开这个连接。这里隐含的一个逻辑就是，连接被断开的时候，会自动回滚这个连接里面正在执行的线程，也就释放了id=1上的行锁。

第二类：查询慢

经过了重重封“锁”，我们再来看看一些查询慢的例子。

先来看一条你一定知道原因的SQL语句：

```
mysql> select * from t where c=50000 limit 1;
```

由于字段c上没有索引，这个语句只能走id主键顺序扫描，因此需要扫描5万行。

作为确认，你可以看一下慢查询日志。注意，这里为了把所有语句记录到slow log里，我在连接后先执行了 set long_query_time=0，将慢查询日志的时间阈值设置为0。

```
# Query_time: 0.011543 Lock_time: 0.000104 Rows_sent: 1 Rows_examined: 50000
SET timestamp=1544723147;
select * from t where c=50000 limit 1;
```

图11 全表扫描5万行的slow log

Rows_examined显示扫描了50000行。你可能会说，不是很慢呀，11.5毫秒就返回了，我们线上一般都配置超过1秒才算慢查询。但你要记住：坏查询不一定是慢查询。我们这个例子里面只有10万行记录，数据量大起来的话，执行时间就线性涨上去了。

扫描行数多，所以执行慢，这个很好理解。

但是接下来，我们再看一个只扫描一行，但是执行很慢的语句。

如图12所示，是这个例子的slow log。可以看到，执行的语句是

```
mysql> select * from t where id=1;
```

虽然扫描行数是1，但执行时间却长达800毫秒。

```
# User@Host: root[root] @ localhost [127.0.0.1] Id:      5
# Query_time: 0.804400 Lock_time: 0.000205 Rows_sent: 1 Rows_examined: 1
SET timestamp=1544728393;
```

图12 扫描一行却执行得很慢

是不是有点奇怪呢，这些时间都花在哪里了？

如果我把这个slow log的截图再往下拉一点，你可以看到下一个语句，select * from t where id=1 lock in share mode，执行时扫描行数也是1行，执行时间是0.2毫秒。

```
# Query_time: 0.000258 Lock_time: 0.000132 Rows_sent: 1 Rows_examined: 1
SET timestamp=1544728398;
select * from t where id=1 lock in share mode;
```

图 13 加上lock in share mode的slow log

看上去是不是更奇怪了？按理说lock in share mode还要加锁，时间应该更长才对啊。

可能有的同学已经有答案了。如果你还没有答案的话，我再给你一个提示信息，图14是这两个

语句的执行输出结果。

```
mysql> select * from t where id=1;
+----+---+
| id | c   |
+----+---+
| 1  | 1   |
+----+---+
1 row in set (0.81 sec)

mysql> select * from t where id=1 lock in share mode;
+----+-----+
| id | c       |
+----+-----+
| 1  | 1000001 |
+----+-----+
1 row in set (0.00 sec)
```

图14 两个语句的输出结果

第一个语句的查询结果里c=1，带lock in share mode的语句返回的是c=1000001。看到这里应该有更多的同学知道原因了。如果你还是没有头绪的话，也别着急。我先跟你说明一下复现步骤，再分析原因。

session A	session B
start transaction with consistent snapshot;	
	update t set c=c+1 where id=1; //执行100万次
select * from t where id=1;	
select * from t where id=1 lock in share mode;	

图15 复现步骤

你看到了，session A先用start transaction with consistent snapshot命令启动了一个事务，之后session B才开始执行update语句。

session B执行完100万次update语句后，id=1这一行处于什么状态呢？你可以从图16中找到答案。

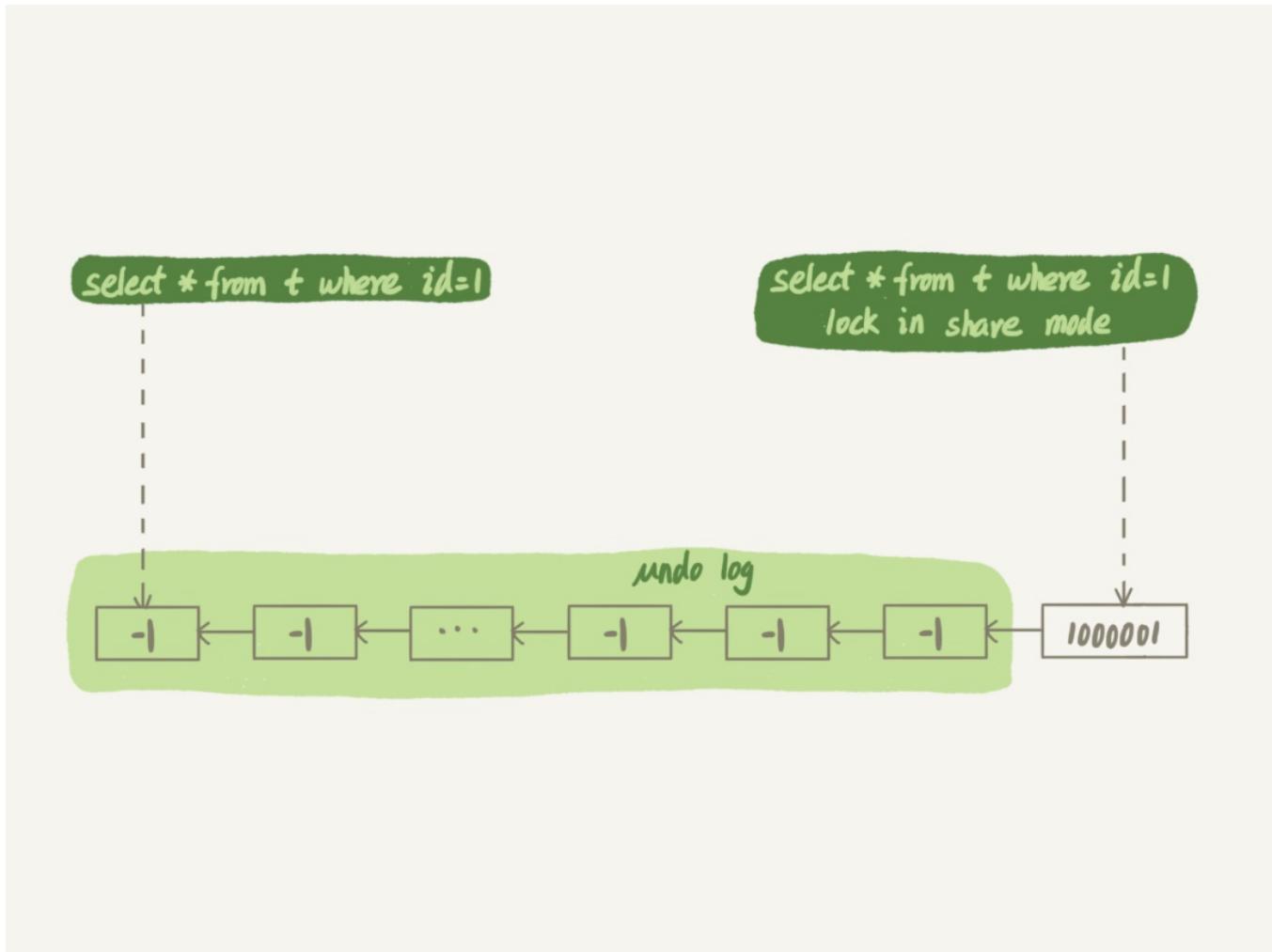


图16 `id=1`的数据状态

session B更新完100万次，生成了100万个回滚日志(undo log)。

带`lock in share mode`的SQL语句，是当前读，因此会直接读到`1000001`这个结果，所以速度很快；而`select * from t where id=1`这个语句，是一致性读，因此需要从`1000001`开始，依次执行`undo log`，执行了100万次以后，才将`1`这个结果返回。

注意，`undo log`里记录的其实是“把2改成1”，“把3改成2”这样的操作逻辑，画成减1的目的是方便你看图。

小结

今天我给你举了一个简单的表上，执行“查一行”，可能出现的被锁住和执行慢的例子。这其中涉及到了表锁、行锁和一致性读的概念。

在实际使用中，碰到的场景会更复杂。但大同小异，你可以按照我在文章中介绍的定位方法，来定位并解决问题。

最后，我给你留一个问题吧。

我们在举例加锁读的时候，用的是这个语句，`select * from t where id=1 lock in share mode`。由

于**id**上有索引，所以可以直接定位到**id=1**这一行，因此读锁也是只加在了这一行上。

但如果是下面的**SQL**语句，

```
begin;  
select * from t where c=5 for update;  
commit;
```

这个语句序列是怎么加锁的呢？加的锁又是什么时候释放呢？

你可以把你的观点和验证方法写在留言区里，我会在下一篇文章的末尾给出我的参考答案。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期间题时间

在上一篇文章最后，我留给你的问题是，希望你可以分享一下之前碰到过的、与文章中类似的场景。

@封建的风 提到一个有趣的场景，值得一说。我把他的问题重写一下，表结构如下：

```
mysql> CREATE TABLE `table_a` (  
    `id` int(11) NOT NULL,  
    `b` varchar(10) DEFAULT NULL,  
    PRIMARY KEY (`id`),  
    KEY `b` (`b`)  
) ENGINE=InnoDB;
```

假设现在表里面，有100万行数据，其中有10万行数据的**b**的值是'1234567890'，假设现在执行语句是这么写的：

```
mysql> select * from table_a where b='1234567890abcd';
```

这时候，MySQL会怎么执行呢？

最理想的情况是，MySQL看到字段**b**定义的是**varchar(10)**，那肯定返回空呀。可惜，MySQL并没有这么做。

那要不，就是把'1234567890abcd'拿到索引里面去做匹配，肯定也没能够快速判断出索引树**b**上并没有这个值，也很快就能返回空结果。

但实际上，MySQL也不是这么做的。

这条SQL语句的执行很慢，流程是这样的：

1. 在传给引擎执行的时候，做了字符截断。因为引擎里面这个行只定义了长度是10，所以只截了前10个字节，就是'1234567890'进去做匹配；
2. 这样满足条件的数据有10万行；
3. 因为是select *， 所以要做10万次回表；
4. 但是每次回表以后查出整行，到server层一判断，b的值都不是'1234567890abcd'；
5. 返回结果是空。

这个例子，是我们文章内容的一个很好的补充。虽然执行过程中可能经过函数操作，但是最终在拿到结果后，server层还是要做一轮判断的。

评论区留言点赞板：

@赖阿甘 提到了等号顺序问题，时间上MySQL优化器执行过程中，where 条件部分，`a=b`和`b=a`的写法是一样的。

@沙漠里的骆驼 提到了一个常见的问题。相同的模板语句，但是匹配行数不同，语句执行时间相差很大。这种情况，在语句里面有`order by`这样的操作时会更明显。

@Justin 回答了我们正文中的问题，如果`id` 的类型是整数，传入的参数类型是字符串的时候，可以用上索引。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。



某、人

最近几张干货越来越多了,很实用,收获不少.先回答今天的问题

版本5.7.13

rc模式下:

session 1:

`begin;`

`select * from t where c=5 for update;`

session 2:

`delete from t where c=10 --等待`

session 3:

`insert into t values(100001,8) --成功`

session 1:

`commit`

session 2:事务执行成功

rr模式下:

`begin;`

`select * from t where c=5 for update;`

session 2:

`delete from t where c=10 --等待`

session 3:

`insert into t values(100001,8) --等待`

session 1:

`commit`

session 2:事务执行成功

session 3: 事务执行成功

从上面这两个简单的例子,可以大概看出上锁的流程.

不管是rr模式还是rc模式,这条语句都会先在server层对表加上MDL S锁,然后进入到引擎层。

rc模式下,由于数据量不大只有10W。通过实验可以证明session 1上来就把该表的所有行都锁住了。

导致其他事务要对该表的所有现有记录做更新,是阻塞状态。为什么insert又能成功?

说明rc模式下for update语句没有上gap锁,所以不阻塞insert对范围加插入意向锁,所以更新成功。

session 1commit后,session 2执行成功。表明所有行的X锁是在事务提交完成以后才释放。

rr模式下,session 1和session 2与rc模式下都一样,说明rr模式下也对所有行上了X锁。

唯一的区别是insert也等待了,是因为rr模式下对没有索引的更新,聚簇索引上的所有记录,都被加上了X锁。其次,聚簇索引每条记录间的间隙(GAP),也同时被加上了GAP锁。由于gap锁阻

塞了insert要加的插入意向锁,导致insert也处于等待状态。只有当session 1 commit完成以后。session 1上的所有锁才会释放,S2,S3执行成功

由于例子中的数据量还比较小,如果数据量达到千万级别,就比较直观的能看出,上锁是逐行上锁的一个过程.扫描一条上一条,直到所有行扫描完,rc模式下对所有行上x锁。rr模式下不仅对所有行上X锁,还对所有区间上gap锁.直到事务提交或者回滚完成后,上的锁才会被释放。

2018-12-26

| 作者回复

分析得非常好。

两个模式下,各增加一个session 4 : update t set c=100 where id=10看看哦

基本就全了!!

2018-12-26



薛畅

9

回来老师的问题:

在 **Read Committed** 隔离级别下,会锁上聚簇索引中的所有记录;

在 **Repeatable Read** 隔离级别下,会锁上聚簇索引中的所有记录,并且会锁上聚簇索引内的所有 **GAP**;

在上面两个隔离级别的情况下,如果设置了 `innodb_locks_unsafe_for_binlog` 开启 **semi-consistent read** 的话,对于不满足查询条件的记录,MySQL 会提前放锁,不过加锁的过程是不可避免的。

2018-12-26



似水流年

4

请问老师,为什么`select blocking_pid from sys.schema_table_lock_waits`;查不到mdl锁的进程id,显示为空。

2018-12-28



沙漠里的骆驼

4

@高枕

这里有些资料提供给你参考:

1. 何登成的技术博客: 加锁分析 <http://hedengcheng.com/?p=771>

2. 锁的常见种类: <http://www.aneastone.com/archives/2017/11/solving-dead-locks-two.html>

2018-12-26



尘封

3

课后问题: d这一列不存在,但是还是要加MDL锁,释放时间应该是事务提交时。

2018-12-26

| 作者回复

抱歉,是要写成`where c=5`,发起堪误了

2018-12-26



尘封

3



工上



老师，有没有遇到过**select**语句一直处于**killed**状态的情况？

2018-12-26

| 作者回复

有，这个是在后面的文章中会用到的例子

2018-12-26



蠢蠢欲动的腹肌



老师，您好

我的**mysql**版本**5.7.24**，尝试的时候发现了如下问题

锁住了表T

mysql> lock table T write;

Query OK, 0 rows affected (0.00 sec)

另一个**terminal**查询时被阻塞，但是查不到**blocking_pid**，这是什么情况呢

mysql> select blocking_pid from sys.schema_table_lock_waits;

Empty set (0.00 sec)

ps:发现查询**schema_table_lock_waits**表与**lock table**的语句不能放在一个**terminal**执行，否则会报

Table 'schema_table_lock_waits' was not locked with LOCK TABLES

自行尝试的同学要注意下，老师有空的话也可以帮看看为什么。。。

2018-12-28



小李子



老师，为什么**session B**执行了**select in share mode**，在等行锁的时候，**session C**执行**select * from sys.innodb_lock_waits where locked_table='test'.`t`**会报这个错

[Err] 1356 - View 'sys.innodb_lock_waits' references invalid table(s) or column(s) or function(s) or definer/invoker of view lack rights to use them，而超时之后，又可以查了？另外，\G 参数会报语法错误？

2018-12-27



Tony Du



对于课后问题，**select * from t where c=5 for update**,

当级别为**RR**时，因为字段**c**上没有索引，会扫主键索引，这时会把表中的记录都加上**X**锁。同时，因为对于**innodb**来说，当级别为**RR**时，是可以解决幻读的，此时对于每条记录的间隙还要加上**GAP**锁。也就是说，表上每一条记录和每一个间隙都锁上了。

当级别为**RC**时，因为字段**c**上没有索引，会扫主键索引，这时会把表中的记录都加上**X**锁。

另外，之前看过相关文章，**MySQL**在实际实现中有些优化措施，比如当**RC**时，在**MySQL server**过滤条件，发现不满足后，会把不满足条件的记录释放锁（这里就是把**c!=5**的记录释放锁），这里会违背两阶段的约束。当然，之前每条记录的加锁操作还是不能省略的。

还有，对于**semi consistent read**开启的情况下，也会提前释放锁。

2018-12-27



信信



老师你好，图3上方提到MySQL 5.7版本修改了MDL的加锁策略，不能复现第六章的场景。但我认为只要仍然满足：DML操作加MDL读锁，DDL操作加MDL写锁，并且事务提交才释放锁，那么就可以复现啊。。。所以5.7到底是改了什么导致无法复现的呢？

2018-12-27



某、人

2

老师我请教一个问题：

flush tables中**close table**的意思是说的把**open_tables**里的表全部关闭掉？下次如果有关于某张表的操作

又把**frm file**缓存进**Open_table_definitions**,把表名缓存到**open_tables**,还是**open_table**只是一个计数？

不是特别明白**flush table**和打开表是个什么流程

2018-12-26

| 作者回复

Flush tables是会关掉表，然后下次请求重新读表信息的

第一次打开表其实就是**open_table_definitions**，包括读表信息一类的

之后再有查询就是拷贝一个对象，加一个计数这样的

2018-12-26



老杨同志

2

愉快的做一下思考题

begin;

select * from t where c=5 for update;

commit;

历史知识的结论是，**innodb**先锁全表的所有行，返回**server**层，判断**c**是否等于5，然后释放**c!=5**的行锁。

验证方法：

事务A执行 锁住一行**c!=5**的记录 比如**id=3 c=3**

select * from t where id = 3 for update 或者 **update t set c=4 where id =3**

然后启动新事务B执行上面的语句**select * from t where c=5 for update;** 看看有没有被阻塞。

用于判断事务B的语句会不会试图锁不满足条件的记录。

然后把事务A和事务B的执行顺序对调一下，也就是先执行B在执行A。看看有没有阻塞，判断在事务B加锁成功的情况下会不会释放不满足查询条件记录的行锁。

2018-12-26

| 作者回复

思路清晰

隔离级别再愉快地改成RR试试

2018-12-26



小确幸

1

问一下：索引扫描与全表扫描，有什么异同点？

2018-12-26

| 作者回复

一般说全表扫描默认是值“扫描主键索引”

2018-12-26



陈旭

1

老师，最近遇到了一个问题，看您有什么建议。

业务场景是这样的：

1.开启事务

2.在表a插入一条记录

3.在表b更新一条记录

4.在表c更新一条记录

5.提交事务

看程序日志所有sql都没问题（没看数据库日志），但是结果是2的那条插入了，3和4都没更新，这个问题有哪几种情况？

2018-12-26

| 作者回复

这是被别的并发事务又改回去了吗？

要么是update的值跟原值相同

要么是update条件没有匹配到行

额，最好给一下每个语句执行后的affected rows，还有binlog里的日志内容，才好分析

2018-12-26



杰之7

0

通过这一节的阅读学习，老师讲述了一个查询语句被锁住和查询慢的两种情况。

在被锁住中，通过等MDL锁，堵住了select查询语句，可以通过kill掉持有MDL写锁。第二种是flush被别的语句堵住，然后flush堵住select语句。第三种是等行锁，通过sys.innodb_lock_wait查到，然后kill。

在查询慢中，lock in share mode直接读1000001

而select * from t where id=1需要从1000001执行100百万行，所以查询就慢了。

老师昨天给我了学习的建议，每个事例在Mysql中去运行做，有不懂的问老师。真的非常感动，也给我了会一直跟随老师学习的动力和勇气。学了专栏的一半，需要把之前学的内容复习，接下来我会跟进老师的课程同时动手做之前的案例，不懂有我的思考之后，会及时请教老师。

2019-01-21



唐名之

0

show VARIABLES LIKE 'performance%';

performance_schema ON

配置已经是打开的

2019-01-11

| 作者回复

诶。。。那奇怪了

执行

select * from performance_schema.metadata_locks; 看看?

2019-01-11



唐名之

0

环境: mysql-5.7.24

```
show VARIABLES LIKE 'performance%';
performance_schema ON
```

A窗口执行: lock table t WRITE;

B窗口执行: select * from t where id=1;

C窗口执行: show PROCESSLIST;

```
53 slave_user DESKTOP-00HHFO4:63064 Binlog Dump 3027 Master has sent all binlog to slave; waiting for more updates
54 root localhost:64572 Sleep 157
55 root localhost:64573 mysql_action Sleep 158
56 root localhost:64575 mysql_action Sleep 156
57 root localhost:64576 mysql_action Sleep 156
58 root localhost:64577 mysql_action Sleep 156
59 root localhost:64578 mysql_action Sleep 156
60 root localhost:64579 mysql_action Sleep 144
61 root localhost:64581 mysql_action Query 140 Waiting for table metadata lock select * from t where id=1
62 root localhost:64583 mysql_action Query 0 starting
```

show PROCESSLIST

已出现: "Waiting for table metadata" 但这三张表都查不出数据, 求解;

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS;
SELECT * from sys.schema_table_lock_waits;
```

2019-01-11

| 作者回复

SELECT * from sys.schema_table_lock_waits; 是需要配置里面把performance_schema打开的;

前面两个语句是只会显示跟innodb的行锁相关的, 表级的锁不会显示在这两个表

2019-01-11



M

老师讲的很好

2019-01-09

0

| 作者回复

多谢鼓励

看文章的同学都很细致，不敢不认真[]

2019-01-09



alias cd=rm -rf

0

思考题

c无索引x锁应该是锁表。

解锁我觉得应该是**sessionb**的事物提交之后

2019-01-08

| 作者回复

不是锁表哈，**innodb**里面除非明确写**lock table**，不会锁表；

解锁时机对的

2019-01-10



小白帽

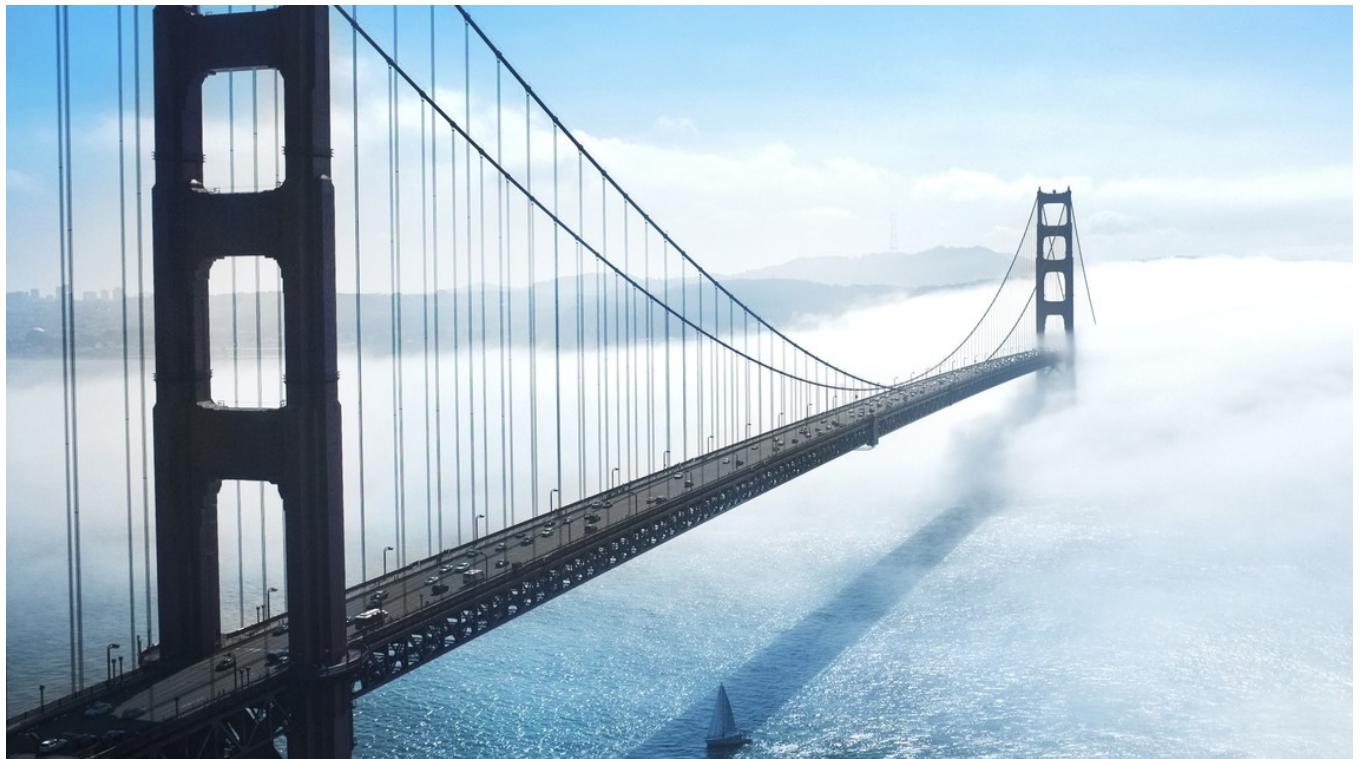
0

涨知识了

2019-01-04

20 | 幻读是什么，幻读有什么问题？

2018-12-28 林晓斌



在上一篇文章最后，我给你留了一个关于加锁规则的问题。今天，我们就从这个问题说起吧。

为了便于说明问题，这一篇文章，我们就先使用一个小一点儿的表。建表和初始化语句如下（为了便于本期的例子说明，我把上篇文章中用到的表结构做了点儿修改）：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `c` (`c`)
) ENGINE=InnoDB;
```

```
insert into t values(0,0,0),(5,5,5),
(10,10,10),(15,15,15),(20,20,20),(25,25,25);
```

这个表除了主键id外，还有一个索引c，初始化语句在表中插入了6行数据。

上期我留给你的问题是，下面的语句序列，是怎么加锁的，加的锁又是什么时候释放的呢？

```

begin;
select * from t where d=5 for update;
commit;

```

比较好理解的是，这个语句会命中**d=5**的这一行，对应的主键**id=5**，因此在**select**语句执行完成后，**id=5**这一行会加一个写锁，而且由于两阶段锁协议，这个写锁会在执行**commit**语句的时候释放。

由于字段**d**上没有索引，因此这条查询语句会做全表扫描。那么，其他被扫描到的，但是不满足条件的**5**行记录上，会不会被加锁呢？

我们知道，**InnoDB**的默认事务隔离级别是可重复读，所以本文接下来没有特殊说明的部分，都是设定在可重复读隔离级别下。

幻读是什么？

现在，我们就来分析一下，如果只在**id=5**这一行加锁，而其他行的不加锁的话，会怎么样。

下面先来看一下这个场景（注意：这是我假设的一个场景）：

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ result: (5,5,5)		
T2		update t set d=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/ result: (0,0,5),(5,5,5)		
T4			insert into t values(1,1,5);
T5	select * from t where d=5 for update; /*Q3*/ result: (0,0,5),(1,1,5),(5,5,5)		
T6	commit;		

图 1 假设只在**id=5**这一行加行锁

可以看到，**session A**里执行了三次查询，分别是**Q1**、**Q2**和**Q3**。它们的**SQL**语句相同，都是**select * from t where d=5 for update**。这个语句的意思你应该很清楚了，查所有**d=5**的行，而且使用的是当前读，并且加上写锁。现在，我们来看一下这三条**SQL**语句，分别会返回什么结果。

1. **Q1**只返回**id=5**这一行；

- 在T2时刻，**session B**把id=0这一行的d值改成了5，因此T3时刻Q2查出来的是id=0和id=5这两行；
- 在T4时刻，**session C**又插入一行(1,1,5)，因此T5时刻Q3查出来的是id=0、id=1和id=5的这三行。

其中，Q3读到id=1这一行的现象，被称为“幻读”。也就是说，幻读指的是一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。

这里，我需要对“幻读”做一个说明：

- 在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的。因此，幻读在“当前读”下才会出现。
- 上面**session B**的修改结果，被**session A**之后的**select**语句用“当前读”看到，不能称为幻读。幻读仅专指“新插入的行”。

如果只从第8篇文章[《事务到底是隔离的还是不隔离的？》](#)我们学到的事务可见性规则来分析的话，上面这三条SQL语句的返回结果都没有问题。

因为这三个查询都是加了**for update**，都是当前读。而当前读的规则，就是要能读到所有已经提交的记录的最新值。并且，**session B**和**session C**的两条语句，执行后就会提交，所以Q2和Q3就是应该看到这两个事务的操作效果，而且也看到了，这跟事务的可见性规则并不矛盾。

但是，这是不是真的没问题呢？

不，这里还真就有问题。

幻读有什么问题？

首先是语义上的。**session A**在T1时刻就声明了，“我要把所有d=5的行锁住，不准别的事务进行读写操作”。而实际上，这个语义被破坏了。

如果现在这样看感觉还不明显的话，我再往**session B**和**session C**里面分别加一条SQL语句，你再看看会出现什么现象。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/		
T2		update t set d=5 where id=0; update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		
T4			insert into t values(1,1,5); update t set c=5 where id=1;
T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 2 假设只在id=5这一行加行锁—语义被破坏

session B的第二条语句update t set c=5 where id=0，语义是“我把id=0、d=5这一行的c值，改成了5”。

由于在T1时刻，session A还只是给id=5这一行加了行锁，并没有给id=0这行加上锁。因此，session B在T2时刻，是可以执行这两条update语句的。这样，就破坏了session A里Q1语句要锁住所有d=5的行的加锁声明。

session C也是一样的道理，对id=1这一行的修改，也是破坏了Q1的加锁声明。

其次，是数据一致性的问题。

我们知道，锁的设计是为了保证数据的一致性。而这个一致性，不止是数据库内部数据状态在此刻的一致性，还包含了数据和日志在逻辑上的一致性。

为了说明这个问题，我给session A在T1时刻再加一个更新语句，即：update t set d=100 where d=5。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ update t set d=100 where d=5;		
T2		update t set d=5 where id=0; update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		
T4			insert into t values(1,1,5); update t set c=5 where id=1;
T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 3 假设只在id=5这一行加行锁—数据一致性问题

update的加锁语义和select ..for update是一致的，所以这时候加上这条update语句也很合理。session A声明说“要给d=5的语句加上锁”，就是为了要更新数据，新加的这条update语句就是把它认为加上了锁的这一行的d值修改成了100。

现在，我们来分析一下图3执行完成后，数据库里会是什么结果。

1. 经过T1时刻，id=5这一行变成(5,5,100)，当然这个结果最终是在T6时刻正式提交的；
2. 经过T2时刻，id=0这一行变成(0,5,5)；
3. 经过T4时刻，表里面多了一行(1,5,5)；
4. 其他行跟这个执行序列无关，保持不变。

这样看，这些数据也没啥问题，但是我们再来看看这时候binlog里面的内容。

1. T2时刻，session B事务提交，写入了两条语句；
2. T4时刻，session C事务提交，写入了两条语句；
3. T6时刻，session A事务提交，写入了update t set d=100 where d=5 这条语句。

我统一放到一起的话，就是这样的：

```
update t set d=5 where id=0; /*(0,0,5)*/
update t set c=5 where id=0; /*(0,5,5)*/
```

```
insert into t values(1,1,5); /*(1,1,5)*/
update t set c=5 where id=1; /*(1,5,5)*/
```

```
update t set d=100 where d=5; /*所有d=5的行, d改成100*/
```

好，你应该看出问题了。这个语句序列，不论是拿到备库去执行，还是以后用binlog来克隆一个库，这三行的结果，都变成了(0,5,100)、(1,5,100)和(5,5,100)。

也就是说，**id=0**和**id=1**这两行，发生了数据不一致。这个问题很严重，是不行的。

到这里，我们再回顾一下，这个数据不一致到底是怎么引入的？

我们分析一下可以知道，这是我们假设“**select * from t where d=5 for update**这条语句只给d=5这一行，也就是**id=5**的这一行加锁”导致的。

所以我们认为，上面的设定不合理，要改。

那怎么改呢？我们把扫描过程中碰到的行，也都加上写锁，再来看看执行效果。

	session A	session B	session C
T1	begin; select * from t where d=5 for update; /*Q1*/ update t set d=100 where d=5;		
T2		update t set d=5 where id=0; (blocked) update t set c=5 where id=0;	
T3	select * from t where d=5 for update; /*Q2*/		
T4			insert into t values(1,1,5); update t set c=5 where id=1;
T5	select * from t where d=5 for update; /*Q3*/		
T6	commit;		

图 4 假设扫描到的行都被加上了行锁

由于**session A**把所有的行都加了写锁，所以**session B**在执行第一个**update**语句的时候就被锁住了。需要等到**T6**时刻**session A**提交以后，**session B**才能继续执行。

这样对于**id=0**这一行，在数据库里的最终结果还是(0,5,5)。在**binlog**里面，执行序列是这样的：

```
insert into t values(1,1,5); /*(1,1,5)*/
update t set c=5 where id=1; /*(1,5,5)

update t set d=100 where d=5; /*所有d=5的行， d改成100*/
update t set d=5 where id=0; /*(0,0,5)*/
update t set c=5 where id=0; /*(0,5,5)
```

可以看到，按照日志顺序执行，**id=0**这一行的最终结果也是(0,5,5)。所以，**id=0**这一行的问题解决了。

但同时你也可以看到，**id=1**这一行，在数据库里面的结果是(1,5,5)，而根据**binlog**的执行结果是(1,5,100)，也就是说幻读的问题还是没有解决。为什么我们已经这么“凶残”地，把所有的记录都上了锁，还是阻止不了**id=1**这一行的插入和更新呢？

原因很简单。在**T3**时刻，我们给所有行加锁的时候，**id=1**这一行还不存在，不存在也就加不上锁。

也就是说，即使把所有的记录都加上锁，还是阻止不了新插入的记录，这也是为什么“幻读”会被单独拿出来解决的原因。

到这里，其实我们刚说明完文章的标题：幻读的定义和幻读有什么问题。

接下来，我们再看看**InnoDB**怎么解决幻读的问题。

如何解决幻读？

现在你知道了，产生幻读的原因是，行锁只能锁住行，但是新插入记录这个动作，要更新的是记录之间的“间隙”。因此，为了解决幻读问题，**InnoDB**只好引入新的锁，也就是间隙锁(**Gap Lock**)。

顾名思义，间隙锁，锁的就是两个值之间的空隙。比如文章开头的表t，初始化插入了6个记录，这就产生了7个间隙。



图 5 表t主键索引上的行锁和间隙锁

这样，当你执行 `select * from t where d=5 for update` 的时候，就不止是给数据库中已有的6个记录加上了行锁，还同时加了7个间隙锁。这样就确保了无法再插入新的记录。

也就是说这时候，在一行行扫描的过程中，不仅将给行加上了行锁，还给行两边的空隙，也加上了间隙锁。

现在你知道了，数据行是可以加上锁的实体，数据行之间的间隙，也是可以加上锁的实体。但是间隙锁跟我们之前碰到过的锁都不太一样。

比如行锁，分成读锁和写锁。下图就是这两种类型行锁的冲突关系。

	读锁	写锁
读锁	兼容	冲突
写锁	冲突	冲突

图6 两种行锁间的冲突关系

也就是说，跟行锁有冲突关系的是“另外一个行锁”。

但是间隙锁不一样，跟间隙锁存在冲突关系的，是“往这个间隙中插入一个记录”这个操作。间隙锁之间都不存在冲突关系。

这句话不太好理解，我给你举个例子：

session A	session B
begin; select * from t where c=7 lock in share mode;	
	begin; select * from t where c=7 for update;

图7 间隙锁之间不互锁

这里**session B**并不会被堵住。因为表t里并没有c=7这个记录，因此**session A**加的是间隙锁(5,10)。而**session B**也是在这个间隙加的间隙锁。它们有共同的目标，即：保护这个间隙，不允许插入值。但，它们之间是不冲突的。

间隙锁和行锁合称**next-key lock**，每个**next-key lock**是前开后闭区间。也就是说，我们的表t初始化以后，如果用**select * from t for update**要把整个表所有记录锁起来，就形成了7个**next-key lock**，分别是(-∞,0]、(0,5]、(5,10]、(10,15]、(15,20]、(20, 25]、(25, +supremum]。

备注：这篇文章中，如果没有特别说明，我们把间隙锁记为开区间，把**next-key lock**记为前开后闭区间。

你可能会问说，这个**supremum**从哪儿来的呢？

这是因为+∞是开区间。实现上，InnoDB给每个索引加了一个不存在的最大值**supremum**，这样才符合我们前面说的“都是前开后闭区间”。

间隙锁和next-key lock的引入，帮我们解决了幻读的问题，但同时也带来了一些“困扰”。

在前面的文章中，就有同学提到了这个问题。我把他的问题转述一下，对应到我们这个例子的表来说，业务逻辑这样的：任意锁住一行，如果这一行不存在的话就插入，如果存在这一行就更新它的数据，代码如下：

```

begin;
select * from t where id=N for update;

/*如果行不存在*/
insert into t values(N,N,N);

/*如果行存在*/
update t set d=N set id=N;

commit;

```

可能你会说，这个不是`insert ...on duplicate key update`就能解决吗？但其实在有多个唯一键的时候，这个方法是不能满足这位提问同学的需求的。至于为什么，我会在后面的文章中再展开说明。

现在，我们就只讨论这个逻辑。

这个同学碰到的现象是，这个逻辑一旦有并发，就会碰到死锁。你一定也觉得奇怪，这个逻辑每次操作前用`for update`锁起来，已经是最严格的模式了，怎么还会有死锁呢？

这里，我用两个session来模拟并发，并假设N=9。

session A	session B
<code>begin;</code> <code>select * from t where id=9 for update;</code>	
	<code>begin;</code> <code>select * from t where id=9 for update;</code>
	<code>insert into t values(9,9,9);</code> (blocked)
<code>insert into t values(9,9,9);</code> (ERROR 1213 (40001): Deadlock found)	

图8 间隙锁导致的死锁

你看到了，其实都不需要用到后面的`update`语句，就已经形成死锁了。我们按语句执行顺序来分析一下：

1. session A 执行`select ...for update`语句，由于`id=9`这一行并不存在，因此会加上间隙锁(5,10)；
2. session B 执行`select ...for update`语句，同样会加上间隙锁(5,10)，间隙锁之间不会冲突，因此这个语句可以执行成功；
3. session B 试图插入一行(9,9,9)，被session A的间隙锁挡住了，只好进入等待；
4. session A试图插入一行(9,9,9)，被session B的间隙锁挡住了。

至此，两个session进入互相等待状态，形成死锁。当然，InnoDB的死锁检测马上就发现了这对死锁关系，让session A的`insert`语句报错返回了。

你现在知道了，间隙锁的引入，可能会导致同样的语句锁住更大的范围，这其实是影响了并发度的。其实，这只是一个简单的例子，在下一篇文章中我们还会碰到更多、更复杂的例子。

你可能会说，为了解决幻读的问题，我们引入了这么一大串内容，有没有更简单一点的处理方法呢。

我在文章一开始就说过，如果没有特别说明，今天和你分析的问题都是在可重复读隔离级别下的，间隙锁是在可重复读隔离级别下才会生效的。所以，你如果把隔离级别设置为读提交的话，就没有间隙锁了。但同时，你要解决可能出现的数据和日志不一致问题，需要把binlog格式设置为row。这，也是现在不少公司使用的配置组合。

前面文章的评论区有同学留言说，他们公司就使用的是读提交隔离级别加binlog_format=row的组合。他曾问他们公司的DBA说，你为什么要这么配置。DBA直接答复说，因为大家都这么用呀。

所以，这个同学在评论区就问说，这个配置到底合不合理。

关于这个问题本身的答案是，如果读提交隔离级别够用，也就是说，业务不需要可重复读的保证，这样考虑到读提交下操作数据的锁范围更小（没有间隙锁），这个选择是合理的。

但其实我想说的是，配置是否合理，跟业务场景有关，需要具体问题具体分析。

但是，如果DBA认为之所以这么用的原因是“大家都这么用”，那就有问题了，或者说，迟早会出问题。

比如说，大家都用读提交，可是逻辑备份的时候，mysqldump为什么要把备份线程设置成可重复读呢？（这个我在前面的文章中已经解释过了，你可以再回顾下第6篇文章[《全局锁和表锁：给表加个字段怎么有这么多阻碍？》](#)的内容）

然后，在备份期间，备份线程用的是可重复读，而业务线程用的是读提交。同时存在两种事务隔离级别，会不会有问题？

进一步地，这两个不同的隔离级别现象有什么不一样的，关于我们的业务，“用读提交就够了”这个结论是怎么得到的？

如果业务开发和运维团队这些问题都没有弄清楚，那么“没问题”这个结论，本身就是有问题的。

小结

今天我们从上一篇文章的课后问题说起，提到了全表扫描的加锁方式。我们发现即使给所有的行都加上行锁，仍然无法解决幻读问题，因此引入了间隙锁的概念。

我碰到过很多对数据库有一定了解的业务开发人员，他们在设计数据表结构和业务SQL语句的时候，对行锁有很准确的认识，但却很少考虑到间隙锁。最后的结果，就是生产库上会经常出现由于间隙锁导致的死锁现象。

行锁确实比较直观，判断规则也相对简单，间隙锁的引入会影响系统的并发度，也增加了锁分析的复杂度，但也有章可循。下一篇篇文章，我就会为你讲解InnoDB的加锁规则，帮你理顺这其中的“章法”。

作为对下一篇课文的预习，我给你留下一个思考题。

session A	session B	session C
begin; select * from t where c>=15 and c<=20 order by c desc for update;		
	insert into t values(11,11,11);	
		insert into t values(6,6,6);

图9 事务进入锁等待状态

如果你之前没有了解过本篇文章的相关内容，一定觉得这三个语句简直是风马牛不相及。但实际上，这里**session B**和**session C**的**insert**语句都会进入锁等待状态。

你可以试着分析一下，出现这种情况的原因是什么？

这里需要说明的是，这其实是在下一篇文章介绍加锁规则后才能回答的问题，是留给你作为预习的，其中**session C**被锁住这个分析是有点难度的。如果你没有分析出来，也不要气馁，我会在下一篇文章和你详细说明。

你也可以说说，你的线上MySQL配置的是什么隔离级别，为什么会这么配置？你有没有碰到什么场景，是必须使用可重复读隔离级别的呢？

你可以把你的碰到的场景和分析写在留言区里，我会在下一篇文章选取有趣的评论跟大家一起分享和分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我们在本文的开头回答了上期问题。有同学的回答中还说明了读提交隔离级别下，在语句执行完成后，是只有行锁的。而且语句执行完成后，InnoDB就会把不满足条件的行行锁去掉。

当然了，`c=5`这一行的行锁，还是会等到**commit**的时候才释放的。

评论区留言点赞板：

@薛畅、@张永志同学给出了正确答案。而且提到了在读提交隔离级别下，是只有行锁的。
@帆帆帆帆帆帆帆帆、@欧阳成对上期的例子做了验证，需要说明一下，需要在启动配置里

面增加`performance_schema=on`, 才能用上这个功能, `performance_schema`库里的表才有数据。



极客时间

MySQL 实战 45 讲

从原理到实战, 丁奇带你搞懂 MySQL

林晓斌 网名丁奇
前阿里资深技术专家



新版升级: 点击「 请朋友读」, 10位好友免费读, 邀请订阅更有**现金奖励**。

精选留言



令狐少侠

5

老师, 今天的文章对我影响很大, 发现之前掌握的知识有些错误的地方, 课后我用你的表结构根据以前不清楚的地方实践了一遍, 现在有两个问题, 麻烦您解答下

1.我在事务1中执行`begin;select * from t where c=5 for update;`事务未提交, 然后事务2中`begin;update t set c=5 where id=0;`执行阻塞, 替换成`update t set c=11 where id=0;`执行不阻塞, 我觉得原因是事务1执行时产生**next-key lock**范围是 $(0,5] \cup (5,10]$ 。我想问下**update set**操作`c=xxx`是否会加锁? 以及加锁的原理。

2.一直以为**gap**只会在二级索引上, 看了你的死锁案例, 发现主键索引上也会有**gap**锁?

2018-12-28

作者回复

1. 好问题。你可以理解为要在索引c上插入一个(`c=5,id=0`)这一行, 是落在 $(0,5] \cup (5,10]$ 里面的, 1可以对吧

2. 嗯, 主键索引的间隙上也要有**Gap lock**保护的

2018-12-28



xuery

0

老师之前的留言说错了, 重新梳理下:

图8：间隙锁导致的死锁；我把`innodb_locks_unsafe_for_binlog`设置为1之后，`session B`并不会`blocked`，`session A insert`会阻塞住，但是不会提示死锁；然后`session B`提交执行成功，`session A`提示主键冲突

这个是因为将`innodb_locks_unsafe_for_binlog`设置为1之后，什么原因造成的？

2019-01-28

| 作者回复

对，`innodb_locks_unsafe_for_binlog`这个参数就是这个意思“不加gap lock”，

这个已经要被废弃了（8.0就没有了），所以不建议设置哈，容易造成误会。

如果真的要去掉gap lock，可以考虑改用RC隔离级别+`binlog_format=row`

2019-02-01



薛畅

8

可重复读隔离级别下，经试验：

`SELECT * FROM t where c>=15 and c<=20 for update;` 会加如下锁：

next-key lock:(10, 15], (15, 20]

gap lock:(20, 25)

`SELECT * FROM t where c>=15 and c<=20 order by c desc for update;` 会加如下锁：

next-key lock:(5, 10], (10, 15], (15, 20]

gap lock:(20, 25)

`session C` 被锁住的原因就是根据索引 `c` 逆序排序后多出的 next-key lock:(5, 10]

同时我有个疑问：加不加 next-key lock:(5, 10] 好像都不会影响到 `session A` 可重复读的语义，那么为什么要加这个锁呢？

2018-12-29

| 作者回复

是的，这个其实就是为啥总结规则有点麻烦，有时候只是因为代码是这么写的

2018-12-29



杜嘉嘉

7

说真的，这一系列文章实用性真的很强，老师非常负责，想必牵扯到老师大量精力，希望老师再出好文章，谢谢您了，辛苦了

2018-12-28

| 作者回复

精力花了没事，睡一觉醒来还是一条好汉

主要还是得大家有收获，我就值了

2018-12-28

```
insert into t values(0,0,0),(5,5,5),
(10,10,10),(15,15,15),(20,20,20),(25,25,25);
```

运行mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select * from t where c>=15 and c<=20 order by c desc for update;

c 索引会在最右侧包含主键值, c索引的值为(0,0) (5,5) (10,10) (15,15) (20,20) (25,25)

此时c索引上锁的范围其实还要匹配主键值。

思考题答案是, 上限会扫到c索引(20,20)上一个键, 为了防止c为20 主键值小于25 的行插入, 需要锁定(20,20) (25,25) 两者的间隙; 开启另一会话(26,25,25)可以插入, 而(24,25,25)会被堵塞。

下限会扫描到(15,15)的下一个键也就是(10,10), 测试语句会继续扫描一个键就是(5,5), 此时会锁定, (5,5) 到(15,15)的间隙, 由于id是主键不可重复所以下限也是闭区间; 在本例的测试数据中添加(21,25,25)后就可以正常插入(24,25,25)

2018-12-28

作者回复

感觉你下一篇看起来会很轻松了哈

2018-12-28



沉浮

5

通过打印锁日志帮助理解问题

锁信息见括号里的说明。

TABLE LOCK table `guo_test`.`t` trx id 105275 lock mode IX

RECORD LOCKS space id 31 page no 4 n bits 80 index c of table `guo_test`.`t` trx id 105275 lock mode X

Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock, 索引锁c (5, 10])

0: len 4; hex 8000000a; asc ;;

1: len 4; hex 8000000a; asc ;;

Record lock, heap no 5 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock, 索引锁c (10,15])

0: len 4; hex 8000000f; asc ;;

1: len 4; hex 8000000f; asc ;;

Record lock, heap no 6 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock, 索引锁c (15,20])

0: len 4; hex 80000014; asc ;;

1: len 4; hex 80000014; asc ;;

Record lock, heap no 7 PHYSICAL RECORD: n_fields 2; compact format; info bits 0 ----(Next-Key Lock, 索引锁c (20,25])

0: len 4; hex 80000019; asc ;;

1: len 4; hex 80000019; asc ;;

RECORD LOCKS space id 31 page no 3 n bits 80 index PRIMARY of table `guo_test`.`t` trx id 105275 lock_mode X locks rec but not gap

Record lock, heap no 5 PHYSICAL RECORD: n_fields 5; compact format; info bits 0

----(记录锁 锁c=15对应的主键)

0: len 4; hex 8000000f; asc ;;

1: len 6; hex 0000000199e3; asc ;;

2: len 7; hex ca000001470134; asc G 4;;

3: len 4; hex 8000000f; asc ;;

4: len 4; hex 8000000f; asc ;;

Record lock, heap no 6 PHYSICAL RECORD: n_fields 5; compact format; info bits 0

0: len 4; hex 80000014; asc ;;

----(记录锁 锁c=20对应的主键)

1: len 6; hex 0000000199e3; asc ;;

2: len 7; hex ca000001470140; asc G @;;

3: len 4; hex 80000014; asc ;;

4: len 4; hex 80000014; asc ;;

由于字数限制，正序及无排序的日志无法帖出，倒序日志比这两者，多了范围(**Next-Key Lock**，索引锁c [5, 10])，个人理解是，加锁分两次，第一次，即正序的锁，第二次为倒序的锁，即多出的(5,10]在RR隔离级别，

innodb在加锁的过程中会默认向后锁一个记录，加上**Next-Key Lock**,第一次加锁的时候10已经在范围，由于倒序，向后，即向5再加**Next-key Lock**,即多出的(5,10]范围

2018-12-28

| 作者回复

优秀

2018-12-28



慧鑫coming

4

这篇需要多读几遍，again

2018-12-28



往事随风，顺其自然

2

总结：**for update** 是锁住所有行还有间隙锁，但是间隙之间互不冲突，但是互不冲突，为什么插入9这一行会被间隙锁等待，原来没有这一行，这和查询9这一行不是一样？

2018-12-28



en

1

老师您好，我mysql的隔离级别是可重复读，数据是(0,0,0),(5,5,5),(10,10,10),(15,15,15),(20,20,20),(25,25,25)，使用了begin;select * from t where c>=15 and c<=20 order by c desc for update;然后sessionB的11阻塞了，但是(6,6,6)的插入成功了这是什么原因呢？

2018-12-31



郭健

1

老师，想请教您几个问题。1.在第六章MDL锁的时候，您说给大表增加字段和增加索引的时候要小心，之前做过测试，给一个一千万的数据增加索引有时需要40分钟，但是增加索引不会对表增加MDL锁吧。除了增加索引慢，还会对数据库有什么影响吗，我问我们dba，他说就开始和结束的时候上一下锁，没什么影响，我个人是持怀疑态度的。2，老师讲到表锁除了MDL锁，还有显示命令lock table的命令的表锁，老师我可以认为，在mysql中如果不显示使用lock table表锁的话，那么mysql是永远不会使用表锁的，如果锁的条件没有索引，使用的是锁住行锁+间隙控制并发。

2018-12-30

| 作者回复

1. 在锁方面你们dba说的基本是对的。一开始和结束有写锁，执行中间40分钟只有读锁但是1000万的表要做40分钟，可能意味着系统压力大（或者配置偏小），这样可能不是没影响对，比较这个操作还是要吃IO和CPU的

2. �恩，innodb引擎是这样的。

2018-12-30



滔滔

1

老师，听了您的课收获满满～～感谢您的付出！您可不可以分析死锁的时候讲一下如何分析死锁日志，期待～～

2018-12-29

| 作者回复

谢谢你的肯定。

嗯死锁分析会有一篇专门说。

不过你可以提前说一下碰到的疑问

2018-12-29



胡月

1

老师，今天线上遇上了一个死锁的问题，您能帮我分析下吗。

根据前面文章的理解：死锁产生的原因如下

线程1：update语句where c= 1 然后 update语句where c=2

线程2：update语句where c=2然后 update语句where c=1

如果线程1获取c=1的锁，等待c=2的锁，线程2获取了c=2的锁，等待c=1的锁，就会产生死锁。

但是线上的情况是

线程1：update语句where c= 1 然后 update语句where c=2

线程2：update语句where c=1然后 update语句where c=2

按说不会产生死锁啊，因为如果线程1获取了c=1的锁，线程2就阻塞了。线程1执行完之后，线程2执行就可以了死锁日志如下：

(1) TRANSACTION:

TRANSACTION 9418928, ACTIVE 0.088 sec fetching rows
mysql tables in use 1, locked 1
LOCK WAIT 66 lock struct(s), heap size 13864, 8 row lock(s)
LOCK BLOCKING MySQL thread id: 11495130 block 11105198
MySQL thread id 11105198, OS thread handle 0x2b086bf45700, query id 88822589 39.106.161.
89 daogou Searching rows for update
UPDATE union_pid
SET USE_TIMES = USE_TIMES + 1
WHERE PID = 'mm_128160800_40474215_33107450401'
(1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 134 page no 93 n bits 192 index `PRIMARY` of table `shanford`.`union_pid` trx id 9418928 lock_mode X locks rec but not gap waiting
Record lock, heap no 86 PHYSICAL RECORD: n_fields 12; compact format; info bits 0

(2) TRANSACTION:

TRANSACTION 9418929, ACTIVE 0.088 sec fetching rows
mysql tables in use 1, locked 1
280 lock struct(s), heap size 46632, 17 row lock(s), undo log entries 1
MySQL thread id 11495130, OS thread handle 0x2b086be41700, query id 88822594 39.106.161.
89 daogou Searching rows for update
UPDATE union_pid
SET USE_TIMES = USE_TIMES + 1
WHERE PID = '1000501132_0_1432392817'
(2) HOLDS THE LOCK(S):
RECORD LOCKS space id 134 page no 93 n bits 192 index `PRIMARY` of table `shanford`.`union_pid` trx id 9418929 lock_mode X locks rec but not gap
Record lock, heap no 86 PHYSICAL RECORD: n_fields 12; compact format; info bits 0

(2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 134 page no 68 n bits 264 index `PRIMARY` of table `shanford`.`union_pid` trx id 9418929 lock_mode X locks rec but not gap waiting
Record lock, heap no 116 PHYSICAL RECORD: n_fields 12; compact format; info bits 0

WE ROLL BACK TRANSACTION (1)

2018-12-29

| 作者回复

PID是唯一索引吗？ 给一下表结构。这两个语句分别对应的主键**ID**如果单独查出来分别是多少
2018-12-29



高枕

林老师，今天我又回头看第四节 深入浅出谈索引（上），里面有这样一段话：为了让一个查询尽量少地读磁盘，就必须让查询过程访问尽量少的数据块。那么，我们就不应该使用二叉树，

1

而是要使用“N 叉”树。这里，“N 叉”树中的“N”取决于数据块的大小。

我想问的是，

一 mysql是以page为最小单位的，mysql一次磁盘io能只读一个块吗？还是多个块组成的page？

二 若一次只能读一个page，也就是多个块的话，这个N的大小是不是应该取决于page的大小呢？

三 主键索引叶子结点存放的实际数据，应该是通过指针跟叶子结点连接的吗？还是直接存在叶子结点所在的页里吗？

2018-12-29



信信

1

老师你好，如果图1的字段d有索引，按前面说的T1时刻后，只有id等于5这一行加了写锁。那么session B 操作的是id等于0这一行，应该不会被阻断吧？如果没阻断的话，仍然会产生语义问题及数据不一致的情况啊。想不明白。。。

2018-12-29

| 作者回复

如果d有索引，而且写法是d=5，那么其他语句要把其他行的d改成5，也是不行的哦

2018-12-29



某、人

1

按照我的理解select * from t where c>=15 and c<=20 order by c desc for update;

这条语句的加锁顺序的以及范围应该是[25,20],[20,15],[15,10],但是通过实验得出来多了(10,5)gap锁

而且不管是用二级索引还是用主键索引，都会加这段gap锁。

有点不太清楚为什么倒序扫描就需要加上了这段gap锁，目的又是为了什么？

不会气磊，期待老师下一期的答案。』

2018-12-28

| 作者回复

嗯嗯下周一见』

2018-12-28



可凡不凡

1

老师

update tab1 set name =(select name from tab2 where status =2)...

tab2.status 上有二级非唯一索引,rr 隔离级别

上述情况

tab2.id 上的索引会被锁吗？

实际开发看到的死锁情况 是这条语句在等待 s 锁 但是没有 gap 锁，也没有设置 semi-consistent read

2018-12-28

| 作者回复

Tab2满足条件的航上会加读锁

2018-12-28



小新

1



这篇文章真的需要多啃几遍，

2018-12-28

| 作者回复

嗯嗯，而且这篇是下篇的基础

2018-12-28



Justin

1

下一章老师会不会讲走普通索引，锁普通索引的时候，主键索引，以及其他索引的加锁顺序或者规则呢？很是好奇

2018-12-28

| 作者回复

嗯嗯，就是这些内容

这篇文章末尾的问题如果一眼看懂的同学应该看起来就轻松的

2018-12-28



林

0

总结就是并发加可重复读引起了数据不一致，也就是幻读的产生，通过间隙锁解决。

2019-02-01

| 作者回复

0

2019-02-02



胡楚坚

0

我对于左开右闭的意义(如果是数学那肯定造的)一直有点迷糊，闭和开有什么区别？然后自己去搜索下： $(a, b]$ 代表着会锁住a跟b之间，不让插入数据，还会锁住数据b本身，但不会锁住数据a(即开和闭对应着要不要锁住数据本身)。老师，我理解的对吗？

至于为什么左开右闭，说是迎合自增主键特性，这就不是很理解了，希望老师有空能回答下。

2019-01-31

| 作者回复

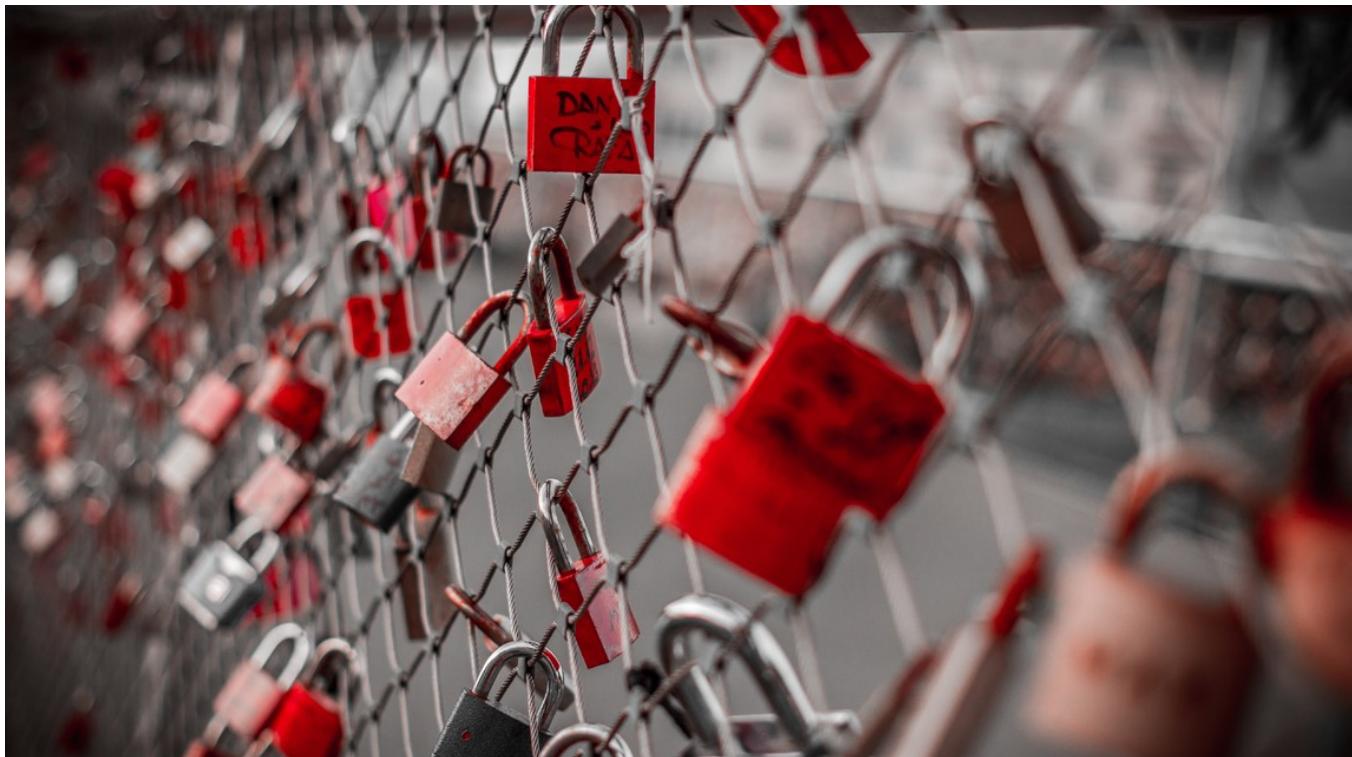
非常正确，就是gap再加上它右边的那个记录。

要让整个区间连续，总要有一边闭区间哈，二选一。然后MySQL一直支持的是升序索引

2019-01-31

21 | 为什么我只改一行的语句，锁这么多？

2018-12-31 林晓斌



在上一篇文章中，我和你介绍了间隙锁和**next-key lock**的概念，但是并没有说明加锁规则。间隙锁的概念理解起来确实有点儿难，尤其在配合上行锁以后，很容易在判断是否会出现锁等待的问题上犯错。

所以今天，我们就先从这个加锁规则开始吧。

首先说明一下，这些加锁规则我没在别的地方看到过有类似的总结，以前我自己判断的时候都是想着代码里面的实现来脑补的。这次为了总结成不看代码的同学也能理解的规则，是我又重新刷了代码临时总结出来的。所以，这个规则有以下两条前提说明：

1. MySQL后面的版本可能会改变加锁策略，所以这个规则只限于截止到现在的最新版本，即5.x系列 \leq 5.7.24，8.0系列 \leq 8.0.13。
2. 如果大家在验证中有发现**bad case**的话，请提出来，我会再补充进这篇文章，使得一起学习本专栏的所有同学都能受益。

因为间隙锁在可重复读隔离级别下才有效，所以本篇文章接下来的描述，若没有特殊说明，默认是可重复读隔离级别。

我总结的加锁规则里面，包含了两个“原则”、两个“优化”和一个“bug”。

1. 原则1：加锁的基本单位是**next-key lock**。希望你还记得，**next-key lock**是前开后闭区间。

2. 原则2：查找过程中访问到的对象才会加锁。
3. 优化1：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。
4. 优化2：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。
5. 一个bug：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。

我还是以上篇文章的表t为例，和你解释一下这些规则。表t的建表语句和初始化语句如下。

```

CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `c` (`c`)
) ENGINE=InnoDB;

insert into t values(0,0,0),(5,5,5),
(10,10,10),(15,15,15),(20,20,20),(25,25,25);

```

接下来的例子基本都是配合着图片说明的，所以我建议你可以对照着文稿看，有些例子可能会“毁三观”，也建议你读完文章后亲手实践一下。

案例一：等值查询间隙锁

第一个例子是关于等值条件操作间隙：

session A	session B	session C
begin; update t set d=d+1 where id=7;		
	insert into t values(8,8,8); (blocked)	
		update t set d=d+1 where id=10; (Query OK)

图1 等值查询的间隙锁

由于表t中没有id=7的记录，所以用我们上面提到的加锁规则判断一下的话：

- 根据原则1，加锁单位是**next-key lock**, session A加锁范围就是(5,10];
- 同时根据优化2，这是一个等值查询(id=7)，而id=10不满足查询条件，**next-key lock**退化成间隙锁，因此最终加锁的范围是(5,10)。

所以，**session B**要往这个间隙里面插入id=8的记录会被锁住，但是**session C**修改id=10这行是可以的。

案例二：非唯一索引等值锁

第二个例子是关于覆盖索引上的锁：

session A	session B	session C
begin; select id from t where c=5 lock in share mode;		
	update t set d=d+1 where id=5; (Query OK)	
		insert into t values(7,7,7); (blocked)

图2 只加在非唯一索引上的锁

看到这个例子，你是不是有一种“该锁的不锁，不该锁的乱锁”的感觉？我们来分析一下吧。

这里**session A**要给索引c上c=5的这一行加上读锁。

- 根据原则1，加锁单位是**next-key lock**，因此会给(0,5]加上**next-key lock**。
- 要注意c是普通索引，因此仅访问c=5这一条记录是不能马上停下来的，需要向右遍历，查到c=10才放弃。根据原则2，访问到的都要加锁，因此要给(5,10]加**next-key lock**。
- 但是同时这个符合优化2：等值判断，向右遍历，最后一个值不满足c=5这个等值条件，因此退化成间隙锁(5,10)。
- 根据原则2，只有访问到的对象才会加锁，这个查询使用覆盖索引，并不需要访问主键索引，所以主键索引上没有加任何锁，这就是为什么**session B**的**update**语句可以执行完成。

但**session C**要插入一个(7,7,7)的记录，就会被**session A**的间隙锁(5,10)锁住。

需要注意，在这个例子中，**lock in share mode**只锁覆盖索引，但是如果是在**for update**就不一样了。执行**for update**时，系统会认为你接下来要更新数据，因此会顺便给主键索引上满足条件的

行加上行锁。

这个例子说明，锁是加在索引上的；同时，它给我们的指导是，如果你要用**lock in share mode**来给行加读锁避免数据被更新的话，就必须得绕过覆盖索引的优化，在查询字段中加入索引中不存在的字段。比如，将**session A**的查询语句改成**select d from t where c=5 lock in share mode**。你可以自己验证一下效果。

案例三：主键索引范围锁

第三个例子是关于范围查询的。

举例之前，你可以先思考一下这个问题：对于我们这个表t，下面这两条查询语句，加锁范围相同吗？

```
mysql> select * from t where id=10 for update;  
mysql> select * from t where id>=10 and id<11 for update;
```

你可能会想，**id**定义为**int**类型，这两个语句就是等价的吧？其实，它们并不完全等价。

在逻辑上，这两条查语句肯定是等价的，但是它们的加锁规则不太一样。现在，我们就让**session A**执行第二个查询语句，来看看加锁效果。

session A	session B	session C
begin; select * from t where id>=10 and id<11 for update;		
	insert into t values(8,8,8); <i>(Query OK)</i> insert into t values(13,13,13); <i>(blocked)</i>	
		update t set d=d+1 where id=15; <i>(blocked)</i>

图3 主键索引上范围查询的锁

现在我们就用前面提到的加锁规则，来分析一下**session A**会加什么锁呢？

- 开始执行的时候，要找到第一个id=10的行，因此本该是next-key lock(5,10]。根据优化1，主键id上的等值条件，退化成行锁，只加了id=10这一行的行锁。
- 范围查找就往后继续找，找到id=15这一行停下来，因此需要加next-key lock(10,15]。

所以，**session A**这时候锁的范围就是主键索引上，行锁id=10和next-key lock(10,15]。这样，**session B**和**session C**的结果你就能理解了。

这里你需要注意一点，首次**session A**定位查找id=10的行的时候，是当做等值查询来判断的，而向右扫描到id=15的时候，用的是范围查询判断。

案例四：非唯一索引范围锁

接下来，我们再看两个范围查询加锁的例子，你可以对照着案例三来看。

需要注意的是，与案例三不同的是，案例四中查询语句的**where**部分用的是字段**c**。

session A	session B	session C
begin; select * from t where c>=10 and c<11 for update;		
	insert into t values(8,8,8); (blocked)	
		update t set d=d+1 where c=15; (blocked)

图4 非唯一索引范围锁

这次**session A**用字段**c**来判断，加锁规则跟案例三唯一的不同是：在第一次用**c=10**定位记录的时候，索引**c**上加了(5,10]这个next-key lock后，由于索引**c**是非唯一索引，没有优化规则，也就是说不会蜕变为行锁，因此最终**sesion A**加的锁是，索引**c**上的(5,10] 和(10,15] 这两个next-key lock。

所以从结果上来看，**session B**要插入 (8,8,8)的这个**insert**语句时就被堵住了。

这里需要扫描到**c=15**才停止扫描，是合理的，因为InnoDB要扫到**c=15**，才知道不需要继续往后找了。

案例五：唯一索引范围锁bug

前面的四个案例，我们已经用到了加锁规则中的两个原则和两个优化，接下来再看一个关于加锁

规则中bug的案例。

session A	session B	session C
begin; select * from t where id>10 and id<=15 for update;		
	update t set d=d+1 where id=20; (blocked)	
		insert into t values(16,16,16); (blocked)

图5 唯一索引范围锁的bug

session A是一个范围查询，按照原则1的话，应该是索引id上只加(10,15]这个next-key lock，并且因为id是唯一键，所以循环判断到id=15这一行就应该停止了。

但是实现上，InnoDB会往前扫描到第一个不满足条件的行为止，也就是id=20。而且由于这是个范围扫描，因此索引id上的(15,20]这个next-key lock也会被锁上。

所以你看到了，session B要更新id=20这一行，是会被锁住的。同样地，session C要插入id=16的一行，也会被锁住。

照理说，这里锁住id=20这一行的行为，其实是没有必要的。因为扫描到id=15，就可以确定不用往后再找了。但实现上还是这么做了，因此我认为这是个bug。

我也曾找社区的专家讨论过，官方bug系统上也有提到，但是并未被verified。所以，认为这是bug这个事儿，也只能算我的一家之言，如果你有其他见解的话，也欢迎你提出来。

案例六：非唯一索引上存在“等值”的例子

接下来的例子，是为了更好地说明“间隙”这个概念。这里，我给表t插入一条新记录。

```
mysql> insert into t values(30,10,30);
```

新插入的这一行c=10，也就是说现在表里有两个c=10的行。那么，这时候索引c上的间隙是什么状态了呢？你要知道，由于非唯一索引上包含主键的值，所以是不可能存在“相同”的两行的。

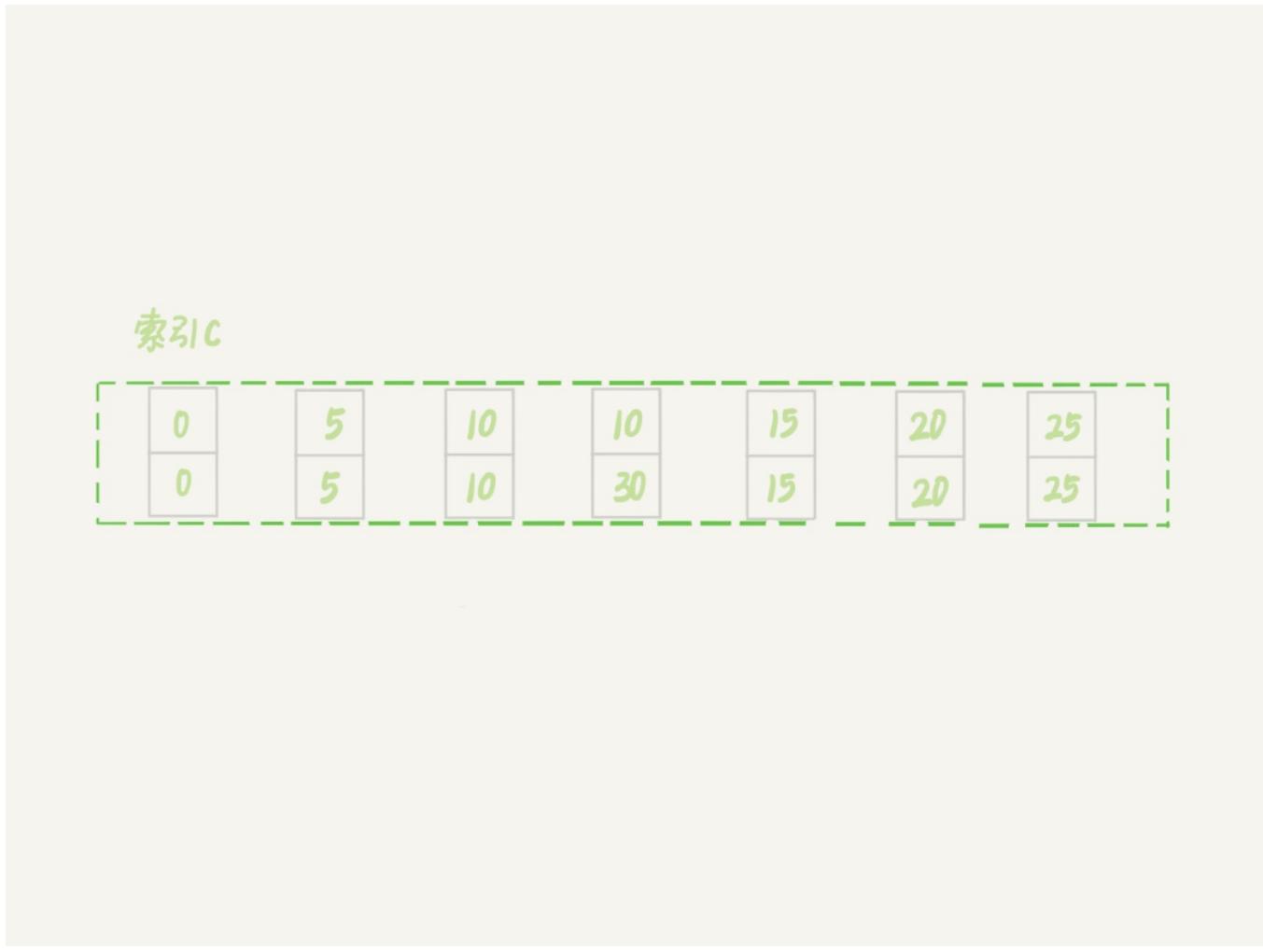


图6 非唯一索引等值的例子

可以看到，虽然有两个 $c=10$ ，但是它们的主键值 id 是不同的（分别是10和30），因此这两个 $c=10$ 的记录之间，也是有间隙的。

图中我画出了索引c上的主键 id 。为了跟间隙锁的开区间形式进行区别，我用 $(c=10,id=30)$ 这样的形式，来表示索引上的一行。

现在，我们来看一下案例六。

这次我们用`delete`语句来验证。注意，`delete`语句加锁的逻辑，其实跟`select ... for update`是类似的，也就是我在文章开始总结的两个“原则”、两个“优化”和一个“bug”。

session A	session B	session C
begin; delete from t where c=10;		
	insert into t values(12,12,12); (blocked)	
		update t set d=d+1 where c=15; (Query OK)

图7 delete 示例

这时， **session A**在遍历的时候，先访问第一个c=10的记录。同样地，根据原则**1**，这里加的是(c=5,id=5)到(c=10,id=10)这个**next-key lock**。

然后， **session A**向右查找，直到碰到(c=15,id=15)这一行，循环才结束。根据优化**2**，这是一个等值查询，向右查找到了不满足条件的行，所以会退化成(c=10,id=10) 到 (c=15,id=15)的间隙锁。

也就是说，这个**delete**语句在索引**c**上的加锁范围，就是下图中蓝色区域覆盖的部分。

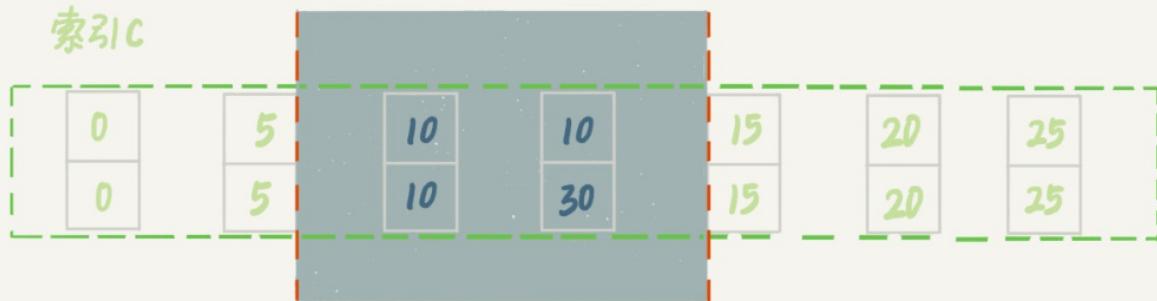


图8 delete加锁效果示例

这个蓝色区域左右两边都是虚线，表示开区间，即($c=5, id=5$)和($c=15, id=15$)这两行上都没有锁。

案例七：limit 语句加锁

例子6也有一个对照案例，场景如下所示：

session A	session B
<pre>begin; delete from t where c=10 limit 2;</pre>	
	<pre>insert into t values(12,12,12); (Query OK)</pre>

图9 limit 语句加锁

这个例子里，session A的delete语句加了 limit 2。你知道表t里 $c=10$ 的记录其实只有两条，因此加不加limit 2，删除的效果都是一样的，但是加锁的效果却不同。可以看到，session B的insert

语句执行通过了，跟案例六的结果不同。

这是因为，案例七里的`delete`语句明确加了`limit 2`的限制，因此在遍历到(`c=10, id=30`)这一行之后，满足条件的语句已经有两条，循环就结束了。

因此，索引`c`上的加锁范围就变成了从 (`c=5,id=5`)到 (`c=10,id=30`)这个前开后闭区间，如下图所示：

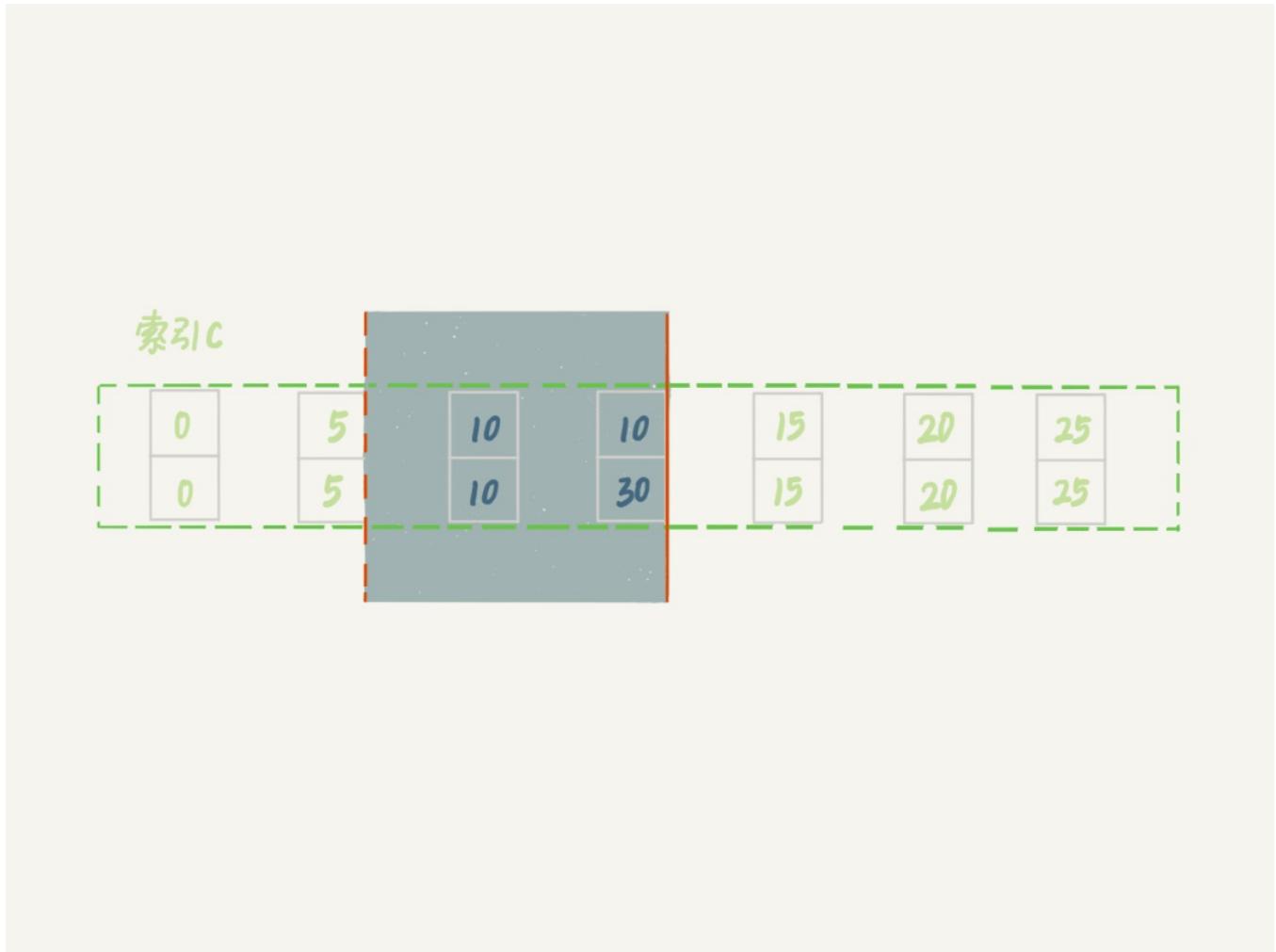


图10 带`limit 2`的加锁效果

可以看到，(`c=10,id=30`)之后的这个间隙并没有在加锁范围里，因此`insert`语句插入`c=12`是可以执行成功的。

这个例子对我们实践的指导意义就是，在删除数据的时候尽量加`limit`。这样不仅可以控制删除数据的条数，让操作更安全，还可以减小加锁的范围。

案例八：一个死锁的例子

前面的例子中，我们在分析的时候，是按照`next-key lock`的逻辑来分析的，因为这样分析比较方便。最后我们再看一个案例，目的是说明：`next-key lock`实际上是间隙锁和行锁加起来的结果。

你一定会疑惑，这个概念不是一开始就说了吗？不要着急，我们先来看下面这个例子：

session A	session B
begin; select id from t where c=10 lock in share mode;	
	update t set d=d+1 where c=10; (blocked)
insert into t values(8,8,8);	
	ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

图11 案例八的操作序列

现在，我们按时间顺序来分析一下为什么是这样的结果。

1. session A 启动事务后执行查询语句加**lock in share mode**，在索引c上加了**next-key lock(5,10]** 和间隙锁**(10,15)**；
2. session B 的**update**语句也要在索引c上加**next-key lock(5,10]**，进入锁等待；
3. 然后**session A**要再插入**(8,8,8)**这一行，被**session B**的间隙锁锁住。由于出现了死锁，InnoDB让**session B**回滚。

你可能会问，**session B**的**next-key lock**不是还没申请成功吗？

其实是这样的，**session B**的“加**next-key lock(5,10]**”操作，实际上分成了两步，先是加**(5,10)**的间隙锁，加锁成功；然后加**c=10**的行锁，这时候才被锁住的。

也就是说，我们在分析加锁规则的时候可以用**next-key lock**来分析。但是要知道，具体执行的时候，是要分成间隙锁和行锁两段来执行的。

小结

这里我再次说明一下，我们上面的所有案例都是在可重复读隔离级别(**repeatable-read**)下验证的。同时，可重复读隔离级别遵守两阶段锁协议，所有加锁的资源，都是在事务提交或者回滚的时候才释放的。

在最后的案例中，你可以清楚地知道**next-key lock**实际上是由间隙锁加行锁实现的。如果切换到

读提交隔离级别(**read-committed**)的话，就好理解了，过程中去掉间隙锁的部分，也就是只剩下行锁的部分。

其实读提交隔离级别在外键场景下还是有间隙锁，相对比较复杂，我们今天先不展开。

另外，在读提交隔离级别下还有一个优化，即：语句执行过程中加上的行锁，在语句执行完成后，就要把“不满足条件的行”上的行锁直接释放了，不需要等到事务提交。

也就是说，读提交隔离级别下，锁的范围更小，锁的时间更短，这也是不少业务都默认使用读提交隔离级别的原因。

不过，我希望你学过今天的课程以后，可以对**next-key lock**的概念有更清晰的认识，并且会用加锁规则去判断语句的加锁范围。

在业务需要使用可重复读隔离级别的时候，能够更细致地设计操作数据库的语句，解决幻读问题的同时，最大限度地提升系统并行处理事务的能力。

经过这篇文章的介绍，你再看一下上一篇文章最后的思考题，再来尝试分析一次。

我把题目重新描述和简化一下：还是我们在文章开头初始化的表t，里面有6条记录，图12的语句序列中，为什么**session B**的**insert**操作，会被锁住呢？

session A	session B
<pre>begin; select * from t where c>=15 and c<=20 order by c desc lock in share mode;</pre>	
	<pre>insert into t values(6,6,6); (blocked)</pre>

图12 锁分析思考题

另外，如果你有兴趣多做一些实验的话，可以设计好语句序列，在执行之前先自己分析一下，然后实际地验证结果是否跟你的分析一致。

对于那些你自己无法解释的结果，可以发到评论区里，后面我争取挑一些有趣的案例在文章中分析。

你可以把你关于思考题的分析写在留言区，也可以分享你自己设计的锁验证方案，我会在下一篇文章的末尾选取有趣的评论跟大家分享。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题，我在本期继续作为课后思考题，所以会在下篇文章再一起公布“答案”。

这里，我展开回答一下评论区几位同学的问题。

- @令狐少侠 说，以前一直认为间隙锁只在二级索引上有。现在你知道了，有间隙的地方就可能有间隙锁。
- @浪里白条 同学问，如果是**varchar**类型，加锁规则是什么样的。
回答：实际上在判断间隙的时候，**varchar**和**int**是一样的，排好序以后，相邻两个值之间就有间隙。
- 有几位同学提到说，上一篇文章自己验证的结果跟案例一不同，就是在**session A**执行完这两个语句：

```
begin;  
select * from t where d=5 for update; /*Q1*/
```

以后，**session B** 的**update** 和**session C**的**insert** 都会被堵住。这是不是跟文章的结论矛盾？

其实不是的，这个例子用的是反证假设，就是假设不堵住，会出现问题；然后，推导出**session A**需要锁整个表所有的行和所有间隙。

评论区留言点赞板：

@某、人、@郭江伟 两位同学尝试分析了上期问题，并给了有启发性的解答。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Lin Xiaobin, a man with glasses and a black shirt, standing with his arms crossed. To his left is the course title "MySQL 实战 45 讲" and a subtitle "从原理到实战，丁奇带你搞懂 MySQL". Above the title is the "极客时间" logo. Below the title, it says "林晓斌 网名丁奇 前阿里资深技术专家". At the bottom, there is a call-to-action: "新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。"



堕落天使

1

老师，您好。假期的没跟上，今天补到了这节课，看了之后有几点不太明白。望能解答一下。

1. 索引c上的锁算不算是行锁。假如索引c上的next-key lock为(0,5] (5,10]，那么5算不算是c上的行锁？
2. 在案例六中，执行“delete from t where c=10;”语句，索引c上的next-key lock是(5,10],(10,10],(10,15)。那么主键索引上的锁是什么呢？是只有行锁，锁住的是(10,10,10)和(30,10,30)两行吗？
3. 也是在案例六中，session A不变，在session B中执行“update t_20 set d=50 where c=5;”、“update t_20 set d=50 where c=15;”、“insert into t_20 values(40,15,40);”均执行成功，但执行“insert into t_20 values(50,5,50);”时，却被阻塞。为什么呢？具体执行语句如下

session A

```
mysql> begin;
mysql> explain delete from t_20 where c=10;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1  | DELETE      | t_20  |          | range | c              | c   | 5       |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> delete from t_20 where c=10;
```

session B

```
mysql> update t_20 set d=50 where c=5;
Query OK, 1 row affected (0.01 sec)

Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> update t_20 set d=50 where c=15;
Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> insert into t_20 values(40,15,40);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> explain insert into t_20 values(50,5,50);
```

```
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1  | INSERT      | t_20  |          | ALL  | c              | c   | 5       |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> insert into t_20 values(50,5,50);
(block)
```

我使用的mysql版本是：5.7.23-0ubuntu0.16.04.1

show variables的结果太多，我截取了一部分，或许对您分析有帮助：

innodb_version 5.7.23

protocol_version 10

slave_type_conversions

tls_version TLSv1,TLSv1.1

version 5.7.23-0ubuntu0.16.04.1

version_comment (Ubuntu)

version_compile_machine x86_64

version_compile_os Linux

2019-01-03

| 作者回复

1. Next-key lock 就是间隙锁+行锁，所以包含=5这一行

2. 对

3. (c=5,id=50)是在这个gap里哦，你试试插入(1,5,50)对比一下。好问题

2019-01-03



张三

25

Happy New Year !这个专栏绝对是极客时间最好我买过最值的专栏。

2018-12-31



约书亚

12

早晨睡不着打开极客时间一看，竟然更新了。今天是周日而且在假期中哎...

2018-12-31

| 作者回复

风雨无阻 节假日不休，包括元旦和春节！

2018-12-31



HuaMax

4

首先老师新年快乐，学习专栏受益良多！

上期间过老师的问题已了解答案，锁是加在索引上的。再尝试回答问题。c上是普通索引，根据原则2，访问到的都要加锁，在查询c>=15这个条件时，在查找到15后加锁（10, 15]，继续往右查找，按理说不会锁住6这个索引值，但查询语句中加了order by c desc，我猜想会优化为使用c<=20这条语句，查找到20后往左查找，这样会访问到15左边的值10，从而加锁（5, 10]，不知我理解对否？

2019-01-01

| 作者回复

新年好

对的！

2019-01-01



郭江伟

4

老师这次的留下的问题，语句跟上次不一样，上期问题语句是`select id from t where c>=15 and c<=20 order by c desc for update;`；这次缺少了`order by c desc`，不加`desc`的话`insert into t values(6,6,6);`不会被堵塞；

根据优化3：索引上的等值查询，在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；

问题中的sql语句加了`desc`，是向左扫描，该优化用不上，所以下限10是闭区间，为了防止`c`为10的行加入，需要锁定到索引`c`键（5,5）

此例中`insert into t values(6,5,6)`会堵塞，`insert into t values(4,5,6)`不会堵塞，

2018-12-31

| 作者回复

嗯你说的对

不过是我少打一个词了，加上去了，要`desc`哦

重新分析下

2018-12-31



undefined

3

遇到一个有趣的问题，在老师的解答下终于弄明白了：

```
CREATE TABLE z (
    id INT PRIMARY KEY AUTO_INCREMENT,
    b INT,
    KEY b(b)
)
ENGINE = InnoDB
DEFAULT CHARSET = utf8;
```

```
INSERT INTO z (id, b)
VALUES (1, 2),
       (3, 4),
       (5, 6),
       (7, 8),
       (9, 10);
```

session A

```
BEGIN;
SELECT *
FROM z
WHERE b = 6 FOR UPDATE;
```

session B

INSERT INTO z VALUES (0, 4);

这里为什么会被锁住

答案比较长，写在我自己的笔记里了，地址是 <https://helloworld.github.io/blog/blog/MySQL/MySQL-%E4%B8%AD%E5%85%B3%E4%BA%8Egap-lock-next-key-lock-%E7%9A%84%E4%BB%80%E4%B8%AA%E9%97%AE%E9%A2%98.html>

大家可以看看

2019-01-07

| 作者回复

好问题，质量很高的笔记

2019-01-10



乾坤

3

您好，关于"优化 2：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。"，我觉得改为"从第一个满足等值条件的索引记录开始向右遍历到第一个不满足等值条件记录，并将第一个不满足等值条件记录上的next-key lock 退化为间隙锁"更明确些

2019-01-01

| 作者回复

感觉没大差别，嗯嗯，理解就好

2019-01-02



Geek_9ca34e

2

老师，你好：

我练习实例的时候发现一个问题：如 案例五：唯一索引范围锁 bug

begin;

select * from t where id>10 and id<=15 for update;

1、执行如上语句加锁范围(10,15]和(15,20];

2、因为10未加锁，所以我单独再开一个连接，执行delete from t where id=10;不会锁等待，能正常删除；

3、但是我再执行insert into t values(10,10,10);语句会等待，无法正常执行；

4、经过分析我发现第一个连接执行的语句的加锁范围已经变成(5,15]和(15,20]，代表锁蔓延了；这是什么原因呢？

2019-01-09

| 作者回复

好问题，我会加到答疑文章中，

Gap是一个动态的概念

2019-01-09



往事随风，顺其自然

2

这和分两步有什么关系？

(5,10]已经是被锁住，分不分两步来加锁，这个间隙和行锁都被锁住了，session b应该是拿不

到锁才对。

2019-01-01



happy涛

1

老师：同上一个问题。还是案例2. `select id from t where c=6 for update;`
ID为[0,9)都不可以添加，包括-1都可以。为啥会锁这么多。而c锁的是[5,10),大于等于5，小于10

2019-01-23

| 作者回复

`select id from t where c=6 for update;`

这个在c上的锁是 (5, 10) 这个间隙

2019-01-23



往事随风，顺其自然

1

session A

`mysql> select * from t where c>=15 and c<=20 order by c desc lock in share mode;`

+-----+-----+

| id | c | d |

+-----+-----+

| 20 | 20 | 20 |

| 15 | 15 | 15 |

+-----+-----+

2 rows in set (0.00 sec)

session b

`mysql> insert into t values(6,6,6);`

Query OK, 1 row affected (0.00 sec)

可以插入成功，没有被锁住

2019-01-01

| 作者回复

`Explain`结果发一下，还有`show variables`结果也发下

2019-01-02



是我的海

0

全是干货赞赞赞，以后出去面试再也不怕面试官装X问锁的问题了

2019-01-31

| 作者回复

一定要低调哈

如果面试的时候能够让大家回答更有底气，那就太好啦

2019-01-31



时隐时现

0

不好意思，这次又来晚了，看这种连载技术文章，和看小说一样，养肥了集中看~~这次的问题如下，希望丁老师有空解答一下。

版本：mysql 5.6.39

```
CREATE TABLE `t` (
`a` int(11) NOT NULL,
`b` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
insert into t values(1,1),(2,2),(3,3),(4,4),(5,5);
```

采用READ-COMMITTED隔离级别

案例1、

session A:

```
begin;
update t set a=6 where b=1;
```

session B:

```
begin;
update t set a=7 where b=2;
```

A和B均能执行成功

问题1：官档上说对于RC且全表扫描的update，先逐行添加行锁然后释放掉不符合where条件的，那么session A成功对(1,1)加锁，理论上session B在扫描(1,1)并尝试加锁时会被阻塞，为何还能执行成功？官档链接：<https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html>

案例2：

session A:

```
begin;
update t set a=6 where b=1;
```

session B:

```
begin;
delete from t where b=2; -- 被阻塞
```

问题2：为何案例1中的session B不会被阻塞，而案例2的却被session A的行数阻塞，update和delete都是全部扫描，难道加锁机制不一样？

2019-01-30

| 作者回复

好问题，在read-committed隔离级别下，update语句有一个“semi-consistent” read优化，

意思是，如果update语句碰到一个已经被锁了的行，会读入最新的版本，然后判断一下是不是满足查询条件，

- a)如果不满足，就直接跳过；
- b)如果满足，才进入锁等待

你的第二个问题：这个策略，只对update有效，delete无效

新春快乐~

2019-02-04



Leon

0

老师，案例八session B的操作语句update t set d = d + 1 where c =10; 由于c是非唯一键索引，锁(5, 10)可以理解，为什么不锁(10,15)呢，不是应该继续向后扫描直到第一个不满足条件的值为止吗

2019-01-29

| 作者回复

好问题，新年快乐

会锁的，只是因为在(5,10]就被锁住了，所以后面的锁加不上去了]

2019-02-01



happy涛

0

老师：

环境同上. QQ466096028

案二：案三也不对。

案例五：b事物也可以执行成功，16, 16, 16我也可以写入，id (10,15) 不可以。

案例六：我没有添加C10, ID30的数。还是用0, 5, 10, 15, 20, 25这几条数据，案例六中的代码执行结果是ID (10, 15), c(5,15) ..

头好疼，感觉理不清，规则太乱了。

2019-01-23

| 作者回复

啊 已经是我简化过的规则了。。需要再理解一下。。

你用session A、sessionB这种模式列一下复现步骤，哪个不清楚的，我们一个个来看吧

2019-01-23



happy涛

0

老师：

案例二：非唯一索引等值锁，这个文章中，事物A加读取之后，按您的文章走，最后结果加的是(5, 10)间隙锁，但我这里为什么插入，c从[0,9)都不能插入。

mysql版本是8.0.12,隔离级别是RR, 表用的是您的例子，数据也是。

2019-01-23

| 作者回复

不是哦

案例二的语句是 where c=5 lock in share mode, 这个在c上的加锁范围是(5,10)

2019-01-23



alias cd=rm -rf

0

思考题：

`order by desc`优化器会向左遍历

- 1、先判断条件 $c \leq 20$, 普通索引等值 $c=20$, 所以next-key-lock: (25, 20]
- 2、20到15, 所以next-key-lock: (20, 15]
- 3、判断 $c >= 15$, 普通索引 $c=15$, 继续向左遍历到 $c=5$ 不符合条件, 并且优化2等值第一个不符合条件的数据降为间隙锁(5,15)
所以锁的范围是(5,15)+[15,20)+[20,25)

2019-01-16

| 作者回复

- 3、判断 $c >= 15$, 普通索引 $c=15$, 继续向左遍历到 $c=5$ 不符合条件, 并且优化2等值第一个不符合条件的数据降为间隙锁(5,15)
所以锁的范围是(5,15)+[15,20)+[20,25)

这个不太对哈（或者说跟我文章里面说的规则不匹配）。

$c >= 15$ 这个条件, 只会向左匹配到 $c=10$ 这个记录,
只是因为next-key lock是前开后闭区间, 所以就是(5,10].

结论的范围也确实是(5,15)+[15,20)+[20,25) □

2019-01-17



J!

0

select max(id) from tb1 和 select id from tb1 order by id desc limit 1; id为主键, 这个两个的加索过程都是一样的吗

2019-01-16

| 作者回复

都不加锁。。

如果说你的是后面加 `for update`, 加索范围一样的

2019-01-16



任洋

0

老师你好, 最近在线上遇到一个问题如下: 执行一个简单的update语句更新数据库, where后面的字段没有索引, 这个字段的数据库中值可能有重复, 在并发的情况下, 会偶发出现数据库死锁的情况。后面通过, 查询出主键, 再通过主键进行更新, 解决了这个问题, 但不明白为什么会出现死锁的情况, 能麻发解释下吗?

2019-01-15

| 作者回复

`update`没索引就是锁住主键索引上所有的行和间隙

锁的内容太多了, 这样确实容易出现死锁哦

2019-01-15



陈

0

老师在案列一中`update t set d=d+1 where id=7` 中`id`是主键也是唯一索引，按优化1应该退化成行锁才对，为什么`insert into t values(8,8,8)`会被锁住，我是那儿理解错了？

2019-01-11

作者回复

这一行存在的时候是行锁，这一行不存在，那就是间隙锁啦。

`insert into t values(8,8,8)`是被主键上(5,10)的间隙锁锁住的

2019-01-11

22 | MySQL有哪些“饮鸩止渴”提高性能的方法？

2019-01-02 林晓斌



不知道你在实际运维过程中有没有碰到这样的情景：业务高峰期，生产环境的MySQL压力太大，没法正常响应，需要短期内、临时性地提升一些性能。

我以前做业务护航的时候，就偶尔会碰上这种场景。用户的开发负责人说，不管你用什么方案，让业务先跑起来再说。

但，如果是无损方案的话，肯定不需要等到这个时候才上场。今天我们就来聊聊这些临时方案，并着重说一说它们可能存在的风险。

短连接风暴

正常的短连接模式就是连接到数据库后，执行很少的SQL语句就断开，下次需要的时候再重连。如果使用的是短连接，在业务高峰期的时候，就可能出现连接数突然暴涨的情况。

我在第1篇文章 [《基础架构：一条SQL查询语句是如何执行的？》](#) 中说过，MySQL建立连接的过程，成本是很高的。除了正常的网络连接三次握手外，还需要做登录权限判断和获得这个连接的数据读写权限。

在数据库压力比较小的时候，这些额外的成本并不明显。

但是，短连接模型存在一个风险，就是一旦数据库处理得慢一些，连接数就会暴涨。`max_connections`参数，用来控制一个MySQL实例同时存在的连接数的上限，超过这个值，系统

就会拒绝接下来的连接请求，并报错提示“**Too many connections**”。对于被拒绝连接的请求来说，从业务角度看就是数据库不可用。

在机器负载比较高的时候，处理现有请求的时间变长，每个连接保持的时间也更长。这时，再有新建连接的话，就可能会超过**max_connections**的限制。

碰到这种情况时，一个比较自然的想法，就是调高**max_connections**的值。但这样做是有风险的。因为设计**max_connections**这个参数的目的是想保护MySQL，如果我们把它改得太大，让更多的连接都可以进来，那么系统的负载可能会进一步加大，大量的资源耗费在权限验证等逻辑上，结果可能是适得其反，已经连接的线程拿不到CPU资源去执行业务的SQL请求。

那么这种情况下，你还有没有别的建议呢？我这里还有两种方法，但要注意，这些方法都是有损的。

第一种方法：先处理掉那些占着连接但是不工作的线程。

max_connections的计算，不是看谁在**running**，是只要连着就占用一个计数位置。对于那些不需要保持的连接，我们可以通过**kill connection**主动踢掉。这个行为跟事先设置**wait_timeout**的效果是一样的。设置**wait_timeout**参数表示的是，一个线程空闲**wait_timeout**这么多秒之后，就会被MySQL直接断开连接。

但是需要注意，在**show processlist**的结果里，踢掉显示为**sleep**的线程，可能是有损的。我们来看下面这个例子。

	session A	session B	session C
T	begin; insert into t values(1,1);	select * from t where id=1;	
T+30s			show processlist;

图1 sleep线程的两种状态

在上面这个例子里，如果断开**session A**的连接，因为这时候**session A**还没有提交，所以MySQL只能按照回滚事务来处理；而断开**session B**的连接，就没什么大影响。所以，如果按照优先级来说，你应该优先断开像**session B**这样的事务外空闲的连接。

但是，怎么判断哪些是事务外空闲的呢？**session C**在T时刻之后的30秒执行**show processlist**，看到的结果是这样的。

ID	User	Host	db	Command	Time	State	Info
4	root	localhost:31998	test	Sleep	30		NULL
5	root	localhost:32114	test	Sleep	30		NULL
6	root	localhost:32166	test	Query	0	starting	show processlist

图2 sleep线程的两种状态， show processlist结果

图中id=4和id=5的两个会话都是Sleep 状态。而要看事务具体状态的话，你可以查information_schema库的innodb_trx表。

```
mysql> select * from information_schema.innodb_trx\G
***** 1. row *****
    trx_id: 1289
    trx_state: RUNNING
    trx_started: 2018-12-23 11:49:17
    trx_requested_lock_id: NULL
    trx_wait_started: NULL
    trx_weight: 2
    trx_mysql_thread_id: 4
    trx_query: NULL
    trx_operation_state: NULL
    trx_tables_in_use: 0
    trx_tables_locked: 1
    trx_lock_structs: 1
    trx_lock_memory_bytes: 1136
    trx_rows_locked: 0
    trx_rows_modified: 1
    trx_concurrency_tickets: 0
    trx_isolation_level: REPEATABLE READ
    trx_unique_checks: 1
    trx_foreign_key_checks: 1
    trx_last_foreign_key_error: NULL
    trx_adaptive_hash_latched: 0
    trx_adaptive_hash_timeout: 0
    trx_is_read_only: 0
    trx_autocommit_non_locking: 0
1 row in set (0.00 sec)
```

图3 从information_schema.innodb_trx查询事务状态

这个结果里，trx_mysql_thread_id=4，表示id=4的线程还处在事务中。

因此，如果是连接数过多，你可以优先断开事务外空闲太久的连接；如果这样还不够，再考虑断开事务内空闲太久的连接。

从服务端断开连接使用的是kill connection + id的命令，一个客户端处于sleep状态时，它的连接被服务端主动断开后，这个客户端并不会马上知道。直到客户端在发起下一个请求的时候，才会收到这样的报错“ERROR 2013 (HY000): Lost connection to MySQL server during query”。

从数据库端主动断开连接可能是有损的，尤其是有的应用端收到这个错误后，不重新连接，而是

直接用这个已经不能用的句柄重试查询。这会导致从应用端看上去，“MySQL一直没恢复”。

你可能觉得这是一个冷笑话，但实际上我碰到过不下10次。

所以，如果你是一个支持业务的DBA，不要假设所有的应用代码都会被正确地处理。即使只是一个断开连接的操作，也要确保通知到业务开发团队。

第二种方法：减少连接过程的消耗。

有的业务代码会在短时间内先大量申请数据库连接做备用，如果现在数据库确认是被连接行为打挂了，那么一种可能的做法，是让数据库跳过权限验证阶段。

跳过权限验证的方法是：重启数据库，并使用`-skip-grant-tables`参数启动。这样，整个MySQL会跳过所有的权限验证阶段，包括连接过程和语句执行过程在内。

但是，这种方法特别符合我们标题里说的“饮鸩止渴”，风险极高，是我特别不建议使用的方案。尤其你的库外网可访问的话，就更不能这么做了。

在MySQL 8.0版本里，如果你启用`-skip-grant-tables`参数，MySQL会默认把`--skip-networking`参数打开，表示这时候数据库只能被本地的客户端连接。可见，MySQL官方对`skip-grant-tables`这个参数的安全问题也很重视。

除了短连接数暴增可能会带来性能问题外，实际上，我们在线上碰到更多的是查询或者更新语句导致的性能问题。其中，查询问题比较典型的有两类，一类是由新出现的慢查询导致的，一类是由QPS（每秒查询数）突增导致的。而关于更新语句导致的性能问题，我会在下一篇文章和你展开说明。

慢查询性能问题

在MySQL中，会引发性能问题的慢查询，大体有以下三种可能：

1. 索引没有设计好；
2. SQL语句没写好；
3. MySQL选错了索引。

接下来，我们就具体分析一下这三种可能，以及对应的解决方案。

导致慢查询的第一种可能是，索引没有设计好。

这种场景一般就是通过紧急创建索引来解决。MySQL 5.6版本以后，创建索引都支持Online DDL了，对于那种高峰期数据库已经被这个语句打挂了的情况，最高效的做法就是直接执行`alter table`语句。

比较理想的是能够在备库先执行。假设你现在的服务是一主一备，主库A、备库B，这个方案的大致流程是这样的：

1. 在备库B上执行 `set sql_log_bin=off`，也就是不写binlog，然后执行`alter table`语句加上索引；
2. 执行主备切换；
3. 这时候主库是B，备库是A。在A上执行 `set sql_log_bin=off`，然后执行`alter table`语句加上索引。

这是一个“古老”的DDL方案。平时在做变更的时候，你应该考虑类似gh-ost这样的方案，更加稳妥。但是在需要紧急处理时，上面这个方案的效率是最高的。

导致慢查询的第二种可能是，语句没写好。

比如，我们犯了在第18篇文章[《为什么这些SQL语句逻辑相同，性能却差异巨大？》](#)中提到的那些错误，导致语句没有使用上索引。

这时，我们可以通过改写SQL语句来处理。MySQL 5.7提供了`query_rewrite`功能，可以把输入的一种语句改写成另外一种模式。

比如，语句被错误地写成了 `select * from t where id + 1 = 10000`，你可以通过下面的方式，增加一个语句改写规则。

```
mysql> insert into query_rewrite.rewrite_rules(pattern, replacement, pattern_database) values ("select * from t where id + 1 = 10000", "select * from t where id = 10000 - 1");
mysql> call query_rewrite.flush_rewrite_rules();
```

这里，`call query_rewrite.flush_rewrite_rules()`这个存储过程，是让插入的新规则生效，也就是我们说的“查询重写”。你可以用图4中的方法来确认改写规则是否生效。

```
mysql> select * from t where id + 1 = 10000;
+----+----+----+
| id | c   | d   |
+----+----+----+
| 9999 | 9999 | 9999 |
+----+----+----+
1 row in set, 1 warning (0.00 sec)

mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message          |
+-----+-----+-----+
| Note  | 1105 | Query 'select * from t where id + 1 = 10000' rewritten to 'select * from t where id = 10000 - 1' by a query rewrite plugin |
+-----+-----+-----+
1 row in set (0.00 sec)
```

图4 查询重写效果

导致慢查询的第三种可能，就是碰上了我们在第10篇文章《MySQL为什么有时候会选错索引？》中提到的情况，MySQL选错了索引。

这时候，应急方案就是给这个语句加上**force index**。

同样地，使用查询重写功能，给原来的语句加上**force index**，也可以解决这个问题。

上面我和你讨论的由慢查询导致性能问题的三种可能情况，实际上出现最多的是前两种，即：索引没设计好和语句没写好。而这两种情况，恰恰是完全可以避免的。比如，通过下面这个过程，我们就可以预先发现问题。

1. 上线前，在测试环境，把慢查询日志（**slow log**）打开，并且把**long_query_time**设置成0，确保每个语句都会被记录入慢查询日志；
2. 在测试表里插入模拟线上的数据，做一遍回归测试；
3. 观察慢查询日志里每类语句的输出，特别留意**Rows_examined**字段是否与预期一致。（我们在前面文章中已经多次用到过**Rows_examined**方法了，相信你已经动手尝试过了。如果还有不明白的，欢迎给我留言，我们一起讨论）。

不要吝啬这段花在上线前的“额外”时间，因为这会帮你省下很多故障复盘的时间。

如果新增的SQL语句不多，手动跑一下就可以。而如果是新项目的话，或者是修改了原有项目的表结构设计，全量回归测试都是必要的。这时候，你需要工具帮你检查所有的SQL语句的返回结果。比如，你可以使用开源工具pt-query-digest(<https://www.percona.com/doc/percona-toolkit/3.0/pt-query-digest.html>)。

QPS突增问题

有时候由于业务突然出现高峰，或者应用程序bug，导致某个语句的**QPS**突然暴涨，也可能导致MySQL压力过大，影响服务。

我之前碰到过一类情况，是由一个新功能的bug导致的。当然，最理想的情况是让业务把这个功能下掉，服务自然就会恢复。

而下掉一个功能，如果从数据库端处理的话，对应于不同的背景，有不同的方法可用。我这里再和你展开说明一下。

1. 一种是由全新业务的bug导致的。假设你的DB运维是比较规范的，也就是说白名单是一个个加的。这种情况下，如果你能够确定业务方会下掉这个功能，只是时间上没那么快，那么就可以从数据库端直接把白名单去掉。
2. 如果这个新功能使用的是单独的数据库用户，可以用管理员账号把这个用户删掉，然后断开

现有连接。这样，这个新功能的连接不成功，由它引发的QPS就会变成0。

3. 如果这个新增的功能跟主体功能是部署在一起的，那么我们只能通过处理语句来限制。这时，我们可以使用上面提到的查询重写功能，把压力最大的SQL语句直接重写成"select 1"返回。

当然，这个操作的风险很高，需要你特别细致。它可能存在两个副作用：

1. 如果别的功能里面也用到了这个SQL语句模板，会有误伤；
2. 很多业务并不是靠这一个语句就能完成逻辑的，所以如果单独把这个语句以select 1的结果返回的话，可能会导致后面的业务逻辑一起失败。

所以，方案3是用于止血的，跟前面提到的去掉权限验证一样，应该是你所有选项里优先级最低的一个方案。

同时你会发现，其实方案1和2都要依赖于规范的运维体系：虚拟化、白名单机制、业务账号分离。由此可见，更多的准备，往往意味着更稳定的系统。

小结

今天这篇文章，我以业务高峰期的性能问题为背景，和你介绍了一些紧急处理的手段。

这些处理手段中，既包括了粗暴地拒绝连接和断开连接，也有通过重写语句来绕过一些坑的方法；既有临时的高危方案，也有未雨绸缪的、相对安全的预案。

在实际开发中，我们也要尽量避免一些低效的方法，比如避免大量地使用短连接。同时，如果你做业务开发的话，要知道，连接异常断开是常有的事，你的代码里要有正确地重连并重试的机制。

DBA虽然可以通过语句重写来暂时处理问题，但是这本身是一个风险高的操作，做好SQL审计可以减少需要这类操作的机会。

其实，你可以看得出来，在这篇文章中我提到的解决方法主要集中在server层。在下一篇文章中，我会继续和你讨论一些跟InnoDB有关的处理方法。

最后，又到了我们的思考题时间了。

今天，我留给你的课后问题是，你是否碰到过，在业务高峰期需要临时救火的场景？你又是怎么处理的呢？

你可以把你的经历和经验写在留言区，我会在下一篇文章的末尾选取有趣的评论跟大家一起分享和分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

前两期我给你留的问题是，下面这个图的执行序列中，为什么**session B**的**insert**语句会被堵住。

session A	session B
begin; select * from t where c>=15 and c<=20 order by c desc lock in share mode;	
	insert into t values(6,6,6); (blocked)

我们用上一篇的加锁规则来分析一下，看看**session A**的**select**语句加了哪些锁：

1. 由于是**order by c desc**，第一个要定位的是索引**c**上“最右边的”**c=20**的行，所以会加上间隙锁**(20,25]**和**next-key lock (15,20]**。
2. 在索引**c**上向左遍历，要扫描到**c=10**才停下来，所以**next-key lock**会加到**(5,10]**，这正是阻塞**session B**的**insert**语句的原因。
3. 在扫描过程中，**c=20**、**c=15**、**c=10**这三行都存在值，由于是**select ***，所以会在主键**id**上加三个行锁。

因此，**session A**的**select**语句锁的范围就是：

1. 索引**c**上 **(5, 25)**;
2. 主键索引上**id=15**、**20**两个行锁。

这里，我再啰嗦下，你会发现我在文章中，每次加锁都会说明是加在“哪个索引上”的。因为，锁就是加在索引上的，这是**InnoDB**的一个基础设定，需要你在分析问题的时候要一直记得。

评论区留言点赞板：

@HuaMax 给出了正确的解释。

@Justin 同学提了个好问题，<=到底是间隙锁还是行锁？其实，这个问题，你要跟“执行过程”配合起来分析。在**InnoDB**要去找“第一个值”的时候，是按照等值去找的，用的是等值判断的规则；找到第一个值以后，要在索引内找“下一个值”，对应于我们规则中说的范围查找。

@信信 提了一个不错的问题，要知道最终的加锁是根据实际执行情况来的。所以，如果一个**select * from ...for update** 语句，优化器决定使用全表扫描，那么就会把主键索引上**next-key lock**全加上。

@nero 同学的问题，提示我需要提醒大家注意，“有行”才会加行锁。如果查询条件没有命中行，那就加**next-key lock**。当然，等值判断的时候，需要加上优化2（即：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。）。

@小李子、@发条橙子同学，都提了很好的问题，这期高质量评论很多，你也都可以去看看。

最后，我要为元旦期间还坚持学习的同学们，点个赞 ^_^

The image shows a promotional banner for a MySQL course. At the top left is the '极客时间' logo. The main title 'MySQL 实战 45 讲' is displayed prominently in large, bold, dark font. Below it, a subtitle reads '从原理到实战，丁奇带你搞懂 MySQL' in a smaller, lighter font. To the right of the text is a portrait of the instructor, Ding Qi, a young man with glasses and a black shirt, standing with his arms crossed. At the bottom left, the author's name '林晓斌' is written, followed by '网名丁奇' and '前阿里资深技术专家'. A call-to-action at the bottom says '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。'.

精选留言

-  某、人 4
- 最近才发生了个案列：
由于一个**delete**大事务导致磁盘空间满了，数据库hang住，连接不上，所以无法**kill**掉该大事务
当时的观察到的现象是：
binlog有一个文件已经达到50多G
ls of | grep delete 该tmp文件100多G
redo log还是只有4个组，每个文件1G
undo log大概有100来G
由于数据库连不上，只有把连接切到从库，**kill**掉主库的进程。过了几分钟，**binlog**文件才缩小为原来的大小。把主库启起来，但是**recovery**非常慢。后面**kill**掉，又以**innodb_force_recovery=3**恢复，**recovery**也是半天没反应。由于这个库也不是重要的库，就把新的主库的备份文件重做了之前的主库，以从库启起来

通过最近的学习+测试分析了下,为什么binlog达到了50多G。tmp文件100多G.
由于binlog_cache不够用,把binlog写进了tmp文件中,binlog文件50多G,说明事务已经执行完成,是binlog在fsync阶段,把空间占满了。fsync并不是一个move而是相当于copy。要等binlog完全落盘以后,才会删除之前的tmp文件。redo log由于是循环写,而且在事务执行过程中,就会把redo log分为mtx落地到磁盘上。所以没有一次性暴增,还是以1G的大小持续写。

我也是后续做测试,观察在事务进行中,redo log文件一直都有变化。binlog没有变化
binlog是在事务执行完以后,才一次性fsync到磁盘

但是为什么recovery=3的情况下,还比较耗时。我估计是之前脏页较多,而redo log又全部被覆盖掉,

需要先通过binlog来恢复redo log,然后再通过redo log来恢复数据页。

请问老师有没有更好的办法来处理这种hang住的情况?

如果在操作系统层面kill掉执行的线程,就好了。

昨天提到的问题3,我也没有测试出来Sending to client这个状态.是之前别人问到的,我也挺懵

2019-01-03

作者回复

先说明下, binlog是没有“恢复redolog”的能力的哈。其它部分分析得很好

Binlog 这么大, 说明是大事务, 崩溃恢复的时候要处理的redolog 很多, 估计耗时间耗在这。

这种磁盘空间满的情况, 以前我的处理方法是把最老的binlog移动到别的盘(如果确定日志已经备份到备份系统了就删掉), 目的是腾出空间让这个事务执行完成。

后面可以考虑这种方案, 强制重启还是有点伤的, 不过核心还是做好监控, 不让出现磁盘100%写满的情况

2019-01-03



Long

9

不是专业DBA, 遇到过几次数据库问题, 有的能解决, 有的好像除了重启或者干等着没啥好办法。

MySQL5.6版本遇到的部分问题:

1. 几个线程处于killed状态一直kill不掉(1天), 然后备份的时候MySQL backup flush all tables with read lock的时候被阻塞, 后面的线程只能等待flush table, kill backup以后也没有办法kill那几个killed状态的语句(processlist显示的killed状态的语句的就是show columns, show create table这样的), 后面没办法, 重启了server。(看到老师后面第25讲有关于kill的解释, 非常期待新知识)

2. 一个非常大(大几百万行)的表truncate, 结果后面所有的线程都阻塞了, 类似于下面这个MySQL bug的场景, 结果就是等这个truncate结束。没有继续干预。

<https://bugs.mysql.com/bug.php?id=80060>

3. 某个新功能上线以后, 一个记录操作人员操作页面操作时间KPI的功能, 由于sql性能不好, 在业务上线跑了3天后数据量增多到临界值, 突然影响了整个系统性能。数据库发现是大量的s

sql执行状态是converting heap to MyISAM, sql写法类似 `select * from table where id(有索引)= xxxx order by yyyy`

DBA以及他们团队要求重启。但是分析了几分钟后提供了几个意见给"DBA", 并解释重启解决不了问题: 首先这个问题重启是解决不了, 因为每次这个sql查询全表, 查询分配的临时表空间不足了, 需要把结果集转到磁盘上, 重启了sql动作没变, 参数没变所以重启解决不了问题。页面查询也没法屏蔽, 页面查询也无法过滤条件,

- (1) 和研发确认后, 表数据删除不影响功能, 只影响客户的KPI报表, 先备份表, 然后删除, 后面等功能修复了再补回去。
- (2) 调整`max_heap_table_size`, `tmp_table_size`, 扩大几倍
- (3) 给这个sql的唯一的一个`order by`字段加个索引。

同时催促研发提供hotfix。最终选择了最简单有效的(1)问题解决, 研发迅速后面也发了hotfix解决了。

4. 某个消费高峰时间段, 高频查询被触发, 一天几十万次执行, 由于存量数据越来越多, 查询性能越来越慢, 主要是索引没有很好规划, 导致CPU资源使用飙升, 后面的sql执行越来越慢。最后尝试了给2个字段添加单独的索引, 解决了50%的问题, 看到执行计划, extra里面, 索引合并使用了intersect, 性能还是慢, 然后立马drop原先的2个单独索引, 创建两个字段的联合索引, 问题解决了。

5. 死锁回滚, 导致的MySQL hang住了, 当时刚入门, 只能简单复现死锁, 没有保留所有的日志, 现在想查也查不到了。。。

感觉大部分都是慢sql和高频事务导致的。

(当然后面的慢sql监控分析, 项目上就很重视了。。)

今天看了这期专栏, 发现5.7的这个功能, `query_rewrite`, 受教了。等我们升到5.7以后, 可以实际操练下。上面的问题3, 也可以用这个功能了(因为是新业务, 新表, 特殊sql, 完全可以起到hotfix的作用)。

请老师帮忙看下上面几次故障是否有更好, 更专业的解决方案。多谢

2019-01-02

作者回复

1. Kill掉备份线程在当时是最好的办法了。不过我之前确实也没碰到过`show create table`不能kill的情况, 我看下代码, 如果能复现出来加入那篇文章中
 2. 噢, 80060这个问题是因为要`truncate`, 所以要回收脏页导致慢, 不过这个问题在5.5.23就优化掉了哦, 看你使用的是5.6, 可能是别的原因。`truncate`如果不是被锁, 而是已经在执行了, 确实还是别做别的事情, 等结束最好;
 3. 这个语句是因为子查询要用临时表, 跟`order by`无关的(你看到的阶段还没开始`order by`操作)。这个语句的临时表都能多到把磁盘打满, 增加`tmp_table_size`是没用的。
- 就是说这三个方法里面2和3其实都无效。你们当时的选择很精准呀。

而且前面提出“重启无效”的这个人值得团队内大力表扬（是不是就是你）

另外这个语句，看着像有机会优化的样子，核心方向是去掉临时表

4. 可以只删掉其中一个独立索引，再加一个联合索引，就是变成(a,b)和(b)这两种索引，也就是把(a)改成(a,b)，这样是加法，相对比较安全。删除索引是一个要很小心的操作，少删一个多一份安全，之后再通过观察索引b的使用情况，确定没必要再删。`interset`确实一般都比较慢。

5. 正常回滚很快的，是不是大事务回滚？这种还是得从消除大事务入手

2019-01-02



某、人

4

老师,我有几个问题：

1. 如果把order by去掉或者order by c asc,往右扫描,为什么没有加[25,30)next-key lock?
2. 执行session A,为什么slow log里的Rows_examined为2?按照答案来讲应该是为3嘛
3. thread states里sending data包括sending data to the client,
另外还有一种state是Sending to client(5.7.8之前叫Writing to net)是writing a packet to the client.
请问针对发送数据给客户端,这两种状态有什么区别?

2019-01-02

| 作者回复

1. Next-key lock是前开后闭区间呀，有扫描到25，所以(20,25]

2. Rows_examined 是server层统计的，这个不满足的值没返回给server

3. 你show processlist 结果发我看下，代码中没搜到

2019-01-02



Tony Du

4

对于上期问题的解答，有一点不是特别理解，

因为order by desc，在索引c上向左遍历，对于(15, 25)这段区间没有问题，
然后，扫描到c=10才停下来，理论上把(10, 15]这个区间锁上就应该是完备的了呀。(5, 10]
这段区间是否锁上对结果应该没有影响呀，为什么会需要(5, 10]这段的next-key lock ?

2019-01-02

| 作者回复

就是这么实现的！

C=10还是要锁的，如果不锁可能被删除

2019-01-02



Tony Du

2

对于上期问题的解答，有一点不是特别理解，

因为order by desc，在索引c上向左遍历，对于(15, 25)这段区间没有问题，
然后，扫描到c=10才停下来，理论上把(10, 15]这个区间锁上就应该是完备的了呀。(5, 10]
这段区间是否锁上对结果应该没有影响呀，为什么会需要(5, 10]这段的next-key lock ?

2019-01-02

作者回复

就是这么实现的

C=10还是要锁的，如果不锁可能被删除

我的回复：

所以，如果把sql改成

```
select * from t where c>=15 and c<=20 order by c asc lock in share mode;
```

那锁的范围就应该是索引c上（10, 25）了吧。

同样查询条件，不同的order顺序，锁的范围不一样，稍微感觉有一点奇怪...

2019-01-03

作者回复

嗯，因为执行索引遍历的顺序不一样，其实锁范围不一样也算合理啦

2019-01-03



Long

1

老师好，看到有的同学在讨论锁的释放问题。

之前分析过一个锁表异常，很多用workbench或者类似客户端的同学可能会遇到，

复现方式：

Step 1: 显示的打开一个事务，或者把autocommit=0，或者mysql workbench中把自动提交的置灰按钮打开以后

Step 2: 执行一个sql（比如，update或者delete之类的），然后sql还没有返回执行结果的中途点击workbench自带的那个stop的红色的按钮。

这个时候很多人可能就不再做其他操作，大多会认为执行已经结束了。但是实际上，锁还在继续锁着的并不会释放。

系统日志记录：

- (1) processlist的状态是sleep, info为null
- (2) innodb_trx的状态是running, trx_query为null
- (3) performance_schema.events_statements_current表中的, sql_text, digest_text: 是有正确的sql的。---这个5.6以后就有了，如果ps打开的话，应该是可以看到的。
`message_text : Query execution was interrupted`
- (4) innodb_locks, lock_waits, 以及show engine innodb status, 只有出现锁等待的时候才会记录，如果只有一个事务的记录行锁，或者表锁，是不会记录的。（不知道是否有参考控制，还是默认的）
- (5) 关于行锁记录数的问题，从测试的结果看，innodb_trx的locked rows，当我点停止的时候，锁定行数保持不变了，当我继续点击执行的时候，锁定记录行数会在之前的记录上向上累加，并不是从0开始。

然后查了audit log以后发现，客户端（mysqlworkbench）送给server端的是KILL QUERY threa

`d_id`, 而不是`Kill thread_id`,

所以MySQL只是终止了事务中的**statement**执行，但是并不会释放锁，因为目前的锁的获取和释放都是基于事务结束的（提交或者回滚）。

这里面关于**kill query/ thread_id**的区别解释

<https://dev.mysql.com/doc/refman/5.6/en/kill.html>

解决方法：

自己解决：`kill` 对应的**thread_id**，或者关闭执行窗口（这个时候会送个`quit`给server端）。

别人解决：有**super**权限的人`kill thread_id`。

关于**kill**的那个文章，其实对所有**DDL, DML**的操作释放过程，还没有全部搞清楚，期待老师的第25讲。

2019-01-02

| 作者回复

总结的非常好，而且现象很全面。

核心的一个点是：`kill query` 只是终止当前执行语句，并不会让事务回滚

2019-01-03



曾剑

1

老师，关于上期遗留问题的解答，我有一点疑惑：

解答中的1中，第一个要定位的是索引 `c` 上“最右边的”`c=20` 的行，为啥只会加上间隙锁（`(20,25)` 和 `next-key lock(15,20]` 呢，为啥不是两个 `next-key lock(15,20]` 和 `(20,25]` 呢？`25` 上的行锁难道是退化的？老师上一篇文章中说到加锁的基本原则中第一点是加锁的基本单位是 `next-key lock`，而退化都是基于索引上的等值查询才会发生呀？盼老师指点迷津。

2019-01-02

| 作者回复

就是优化2，找第一个值的时候是等值查询

2019-01-02



Invictus_CD

0

老师好，这个课后题`c≥15`加锁和上一课的例子4的`c≥10`解释的不太一样啊。例子4的直接在`10`上面加的间隙锁啊，这个为啥要在`5`上面加呢？

2019-02-08

| 作者回复

上一篇的案例4，`session A`的**select**语句没有`order by c desc`

区别就是在“`order by c desc`”上

看一下30篇哈

2019-02-09



刘昆

0

老师你好，上期问题里面我遇到一下问题：

`insert into t values(6,5,6) => block`

insert into t values(4,5,6) => no block

insert into t values(6,4,6) => no block

insert into t values(7,5,6) => block

insert into t values(7,4,6) => no block

根据你的解答，c 上面的 next-key lock 在 (5, 10]，那么上面的情况应该都不会阻塞还对呀？

Server version: 5.7.24-log MySQL Community Server (GPL)

2019-02-02

作者回复

是这样的，我们只是简写成(5,10],

这个是索引c上的next-key lock,

所以这个范围的左边界是 (c=5,id=5), 右边界是(c=10,id=10)

你举例里面，

insert into t values(6,5,6) 是 (c=5, id=6);

insert into t values(7,5,6) 是 (c=5, id=7);

这两个都落在上面的next-key lock的区间，所以是会被锁住的哦

好问题，新年快乐

2019-02-03



Moby

0

谢谢谢谢谢谢老师的回答！“作者回复

这没问题呀

begin; select * from t where c>=15 and c<=20 order by c desc lock in share mode;

锁的范围是这样的：

索引c上，next-key lock: (5, 10],(10,15],(15,20];

索引id上，行锁: id=15和id=20”

不过在文末（二十二：MySQL有哪些“饮鸩止渴”提高性能的方法）上写的是“

因此，session A 的 select 语句锁的范围就是：1.索引c上(5,25); 2. 主键索引上id=10、15、20三个行锁”（写错了吧？）

2019-01-22

作者回复

嗯嗯，你说的对，我这里弄错了，应该是“主键索引上id=15、20两个行锁”

勘误啦 多谢

2019-01-23



Moby

0

丁奇老师好，不好意思，学渣看得比较慢。关于前两期的问题，我有一点没搞懂。就是你说的：“session A 在 select 语句锁的范围是 1.... ; 2.在主键索引上id=10、15、20三个行锁”，经我测试(MySQL版本：5.7.17-log; 隔离级别：可重复读): “session

A: begin; select * from t where c>=15 and c<=20 order by c desc lock in share mode;"、"session B: update t set c=1010 where id=10; Query ok"、"session C: update t set c=1515 where id=15;block..."。即：为什么id=10这一行可以更新数据？而id=15、20这两行更新数据就被阻塞？

2019-01-21

| 作者回复

这没问题呀

begin; select * from t where c>=15 and c<=20 order by c desc lock in share mode;

锁的范围是这样的：

索引c上， next-key lock: (5, 10],(10,15],(15,20];

索引id上， 行锁: id=15和id=20

2019-01-21



unlock

0

老师，对于这句话“锁就是加在索引上的”，如果一个表没有主键、没有索引，还会加锁吗。
如果会加，加到哪

2019-01-18

| 作者回复

InnoDB可不存在“没有索引的表”哦

没有主键，系统会给创建一个的（隐藏的

2019-01-18



往事随风，顺其自然

0

为什么我的电脑上没有慢查询的日志文件，mysql5.7

mysql> show VARIABLES like '%slow%';

Variable_name	Value
log_slow_admin_statements	OFF
log_slow_slave_statements	OFF
slow_launch_time	2
slow_query_log	ON
slow_query_log_file	DESKTOP-76FNKS3-slow.log

DESKTOP-76FNKS3-slow.log 这个文件再本地磁盘找不到

2019-01-04

| 作者回复

Show variables like "output"

2019-01-04



往事随风，顺其自然

0



mysql5.7为什么不存在下面的数据库和表

```
mysql> use query_rewrite;
```

```
ERROR 1049 (42000): Unknown database 'query_rewrite'
```

```
mysql>
```

2019-01-04

| 作者回复

搜一下用法吧

2019-01-04



堕落天使

0

老师，您好：

我引用一下 Ryoma 的留言，如下：

Ryoma

我之前的描述有点问题，其实想问的是：为什么加了 `order by c desc`，第一个定位`c=20`的行，会加上间隙锁 `(20,25)` 和 `next-key lock (15,20]`？

如果没有 `order by c desc`，第一次命中`c=15`时，只会加上 `next-key lock(10,15]`；

而有了 `order by c desc`，我的理解是第一次命中`c=20`只需要加上 `next-key lock (15,20]`

当然最后`(20,25)`还是加上了锁，老师的结论是对的，我也测试过了，但是我不知道如何解释。

唯一能想到的解释是 `order by c desc` 并不会改变优化2这个原则：即等值查询时，会向右遍历且最后一个值不满足等值条件；同时 `order by c desc` 带来一个类似于优化2的向左遍历原则。

进而导致最后的锁范围是`(5,25)`；而没有 `order by c desc` 的范围是`(10,25]`。

2019-01-03

| 作者回复

因为执行`c=20`的时候，由于要 `order by c desc`，就要先找到“最右边第一个`c=20`的行”，这个怎么找呢，只能向右找到`25`，才能知道它左边那个`20`是“最右的`20`”

我的问题是：

1. 按照老师您说的，先找`c=20`，由于是 `order by c desc`，所以要找最右边的`20`，即找到`25`。那如果`c`是唯一索引呢？是不是就不会找到`25`了（是否会加 `(20,25)` 的gap lock）？我把语句改造了一下，“`select * from t_20 where id >= 15 and id<=20 ORDER BY id desc lock in share mode ;`”。发现当 session A 执行完这行语句不提交的时候，session B 执行 “`insert into t_20 values(24,24,24);`” 是阻塞的。也就是说也加了`(20,25)`的间隙锁。这又是为什么呢？

2. 间隙锁本身不冲突，但和插入语句冲突。那么 `delete` 语句呢？

我做了个如下实验（以下语句按时间顺序排序）：

session A

```
begin;
```

```
select * from t_20 where c=10 lock in share mode;
```

session B

```
delete from t_20 where c=15;
```

```
insert into t_20 values(16,16,16);
```

(blocked)

session B 中第一条delete语句执行正常，第二条insert语句被阻塞。

我的分析是： session A在索引c上的锁是：(5,10] (10,15)；当session B把(15,15,15)这条记录删了之后，(10,15)的间隙就不存在了，所以此时session A在索引c上的锁变为：(5,10] (10,20)。这时再在session B中插入(16,16,16)就被阻塞了。这个分析正确吗？

2019-01-04

作者回复

对，我在第30篇会说到这个问题哈

2019-01-10



不二

0

老师，曾剑同学的问题

关于上期遗留问题的解答，我有一点疑惑：

解答中的1中，第一个要定位的是索引 c 上“最右边的” $c=20$ 的行，为啥只会加上间隙锁 (20,25) 和next-key lock(15,20]呢，为啥不是两个next-key lock(15,20]和(20,25]呢？25上的行锁难道是退化的？老师上一篇文章中说到加锁的基本原则中第一点是加锁的基本单位是next-key lock，而退化都是基于索引上的等值查询才会发生呀？盼老师指点迷津。

您给回答是定位到 $c=20$ 的时候，是等值查询，所以加的是(20,25)的间隙锁，25的行锁退化了，那么在上一期中的案例五：唯一索引范围锁 bug，那 $id \leq 15$,不也是先定位到 $id=15$ ，然后向右扫描，那应该也是等值查询，那么应该加的是 (15, 20) 间隙锁，那为啥你说的加的是 (15, 20] ,为啥这个 $id=20$ 的行锁也加上了呢，为啥同样是范围查询，一个行锁退化了，一个没有退化呢，求老师指点迷津

2019-01-04

作者回复

1. 第一次就是找 $c=20$,这个就是一次等值查找

2. 案例5那个，等值查的是 $id=10$,然后向右遍历。这两个，一个是有order by desc,索引的扫描方向不一样，“找第一个”的值也是不一样的

2019-01-04



张永志

0

说一个锁全库(schema)的案例，数据库晚间定时任务执行CTAS操作，由于需要执行十几分钟，导致严重会话阻塞，全库所有表上的增删改查全被阻塞。

后改为先建表再插数解决。

2019-01-04

作者回复

嗯嗯。看来你也是趟过好多坑啦，

CTAS不是好用法[]

2019-01-04



张永志

0

我是从Oracle转到MySQL来的，先接触的Oracle再看MySQL就经常喜欢拿两者对比，包括表数据存储结构，二级索引的异同，redo, binlog, 锁机制，以及默认隔离级别。

研究锁后，根据自己的理解得出一个结论，MySQL默认隔离级别选为RR也是无奈之举！

因为当时binlog还是语句格式，为了保证binlog事务顺序正确就得有gap和next key锁。而对开发人员来说，他们未必清楚事务隔离级别，且大多数开发都是从Oracle转向MySQL的，故果断将隔离级别全部调整为RC。

2019-01-04

作者回复

是的，以前有很多oracle专家，然后大家就觉得RC够用。

不过他们不是“以为够用”，他们是真的分析过业务场景，分析业务的用法，确认够用。这种是很好的实践

2019-01-04



张永志

0

分享一个主从切换时遇到的问题，主从切换前主库要改为只读，设置只读后，show master status发现binlog一直在变化，当时应用没断开。

主库并不是其他库的从库，怎么搞的呢？

检查业务用户权限发现拥有super权限，查看授权语句原来是grant all on *.* to user，这里要说的是*.* 权限就太大了，而且这个也很容易被误解，需要特别注意。

2019-01-04

作者回复

对的，readonly对super无效；

一方面是尽量不要给业务super

一方面你做完readonly还会去确认binlog有没有变，这个意识很好哦

2019-01-04



张永志

0

小系统，昨天一直报CPU使用率高，报警阈值设定为CPU平均使用率0.8。

登录看进程都在执行同一条SQL，活动会话有40个，主机逻辑CPU只有4个，这负载能不高吗？

检查SQL，表很小不到两万行，创建一个复合索引后，负载立刻就消失不见啦！

2019-01-04

作者回复

立竿见影

2019-01-04

23 | MySQL是怎么保证数据不丢的？

2019-01-04 林晓斌



今天这篇文章，我会继续和你介绍在业务高峰期临时提升性能的方法。从文章标题“MySQL是怎么保证数据不丢的？”，你就可以看出来，今天我和你介绍的方法，跟数据的可靠性有关。

在专栏前面文章和答疑篇中，我都着重介绍了WAL机制（你可以再回顾下[第2篇](#)、[第9篇](#)、[第12篇](#)和[第15篇](#)文章中的相关内容），得到的结论是：只要redo log和binlog保证持久化到磁盘，就能确保MySQL异常重启后，数据可以恢复。

评论区有同学又继续追问，redo log的写入流程是怎么样的，如何保证redo log真实地写入了磁盘。那么今天，我们就再一起看看MySQL写入binlog和redo log的流程。

binlog的写入机制

其实，binlog的写入逻辑比较简单：事务执行过程中，先把日志写到binlog cache，事务提交的时候，再把binlog cache写到binlog文件中。

一个事务的binlog是不能被拆开的，因此不论这个事务多大，也要确保一次性写入。这就涉及到了binlog cache的保存问题。

系统给binlog cache分配了一片内存，每个线程一个，参数 binlog_cache_size用于控制单个线程内binlog cache所占内存的大小。如果超过了这个参数规定的大小，就要暂存到磁盘。

事务提交的时候，执行器把binlog cache里的完整事务写入到binlog中，并清空binlog cache。状

态如图1所示。

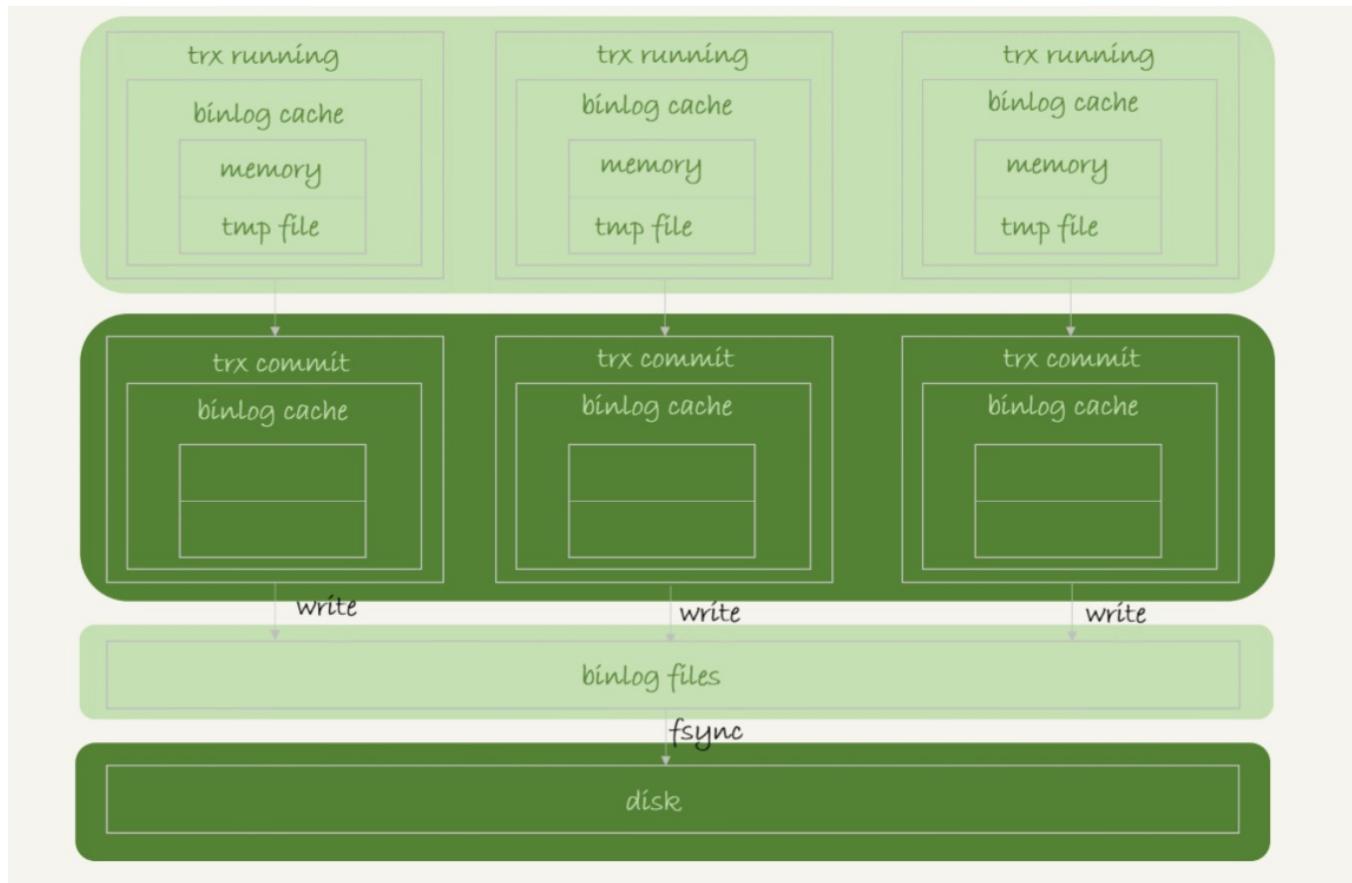


图1 binlog写盘状态

可以看到，每个线程都有自己binlog cache，但是共用同一份binlog文件。

- 图中的**write**，指的就是指把日志写入到文件系统的**page cache**，并没有把数据持久化到磁盘，所以速度比较快。
- 图中的**fsync**，才是将数据持久化到磁盘的操作。一般情况下，我们认为**fsync**才占磁盘的**IOPS**。

write 和 **fsync** 的时机，是由参数 **sync_binlog** 控制的：

1. **sync_binlog=0** 的时候，表示每次提交事务都只**write**，不**fsync**；
2. **sync_binlog=1** 的时候，表示每次提交事务都会执行**fsync**；
3. **sync_binlog=N(N>1)** 的时候，表示每次提交事务都**write**，但累积 **N** 个事务后才**fsync**。

因此，在出现IO瓶颈的场景里，将**sync_binlog** 设置成一个比较大的值，可以提升性能。在实际的业务场景中，考虑到丢失日志量的可控性，一般不建议将这个参数设成0，比较常见的是将其设置为100~1000中的某个数值。

但是，将**sync_binlog** 设置为**N**，对应的风险是：如果主机发生异常重启，会丢失最近**N**个事务的**binlog** 日志。

redo log的写入机制

接下来，我们再说说redo log的写入机制。

在专栏的[第15篇答疑文章](#)中，我给你介绍了redo log buffer。事务在执行过程中，生成的redo log是要先写到redo log buffer的。

然后就有同学问了，redo log buffer里面的内容，是不是每次生成后都要直接持久化到磁盘呢？

答案是，不需要。

如果事务执行期间MySQL发生异常重启，那这部分日志就丢了。由于事务并没有提交，所以这时日志丢了也不会有损失。

那么，另外一个问题是，事务还没提交的时候，redo log buffer中的部分日志有没有可能被持久化到磁盘呢？

答案是，确实会有。

这个问题，要从redo log可能存在的三种状态说起。这三种状态，对应的就是图2 中的三个颜色块。

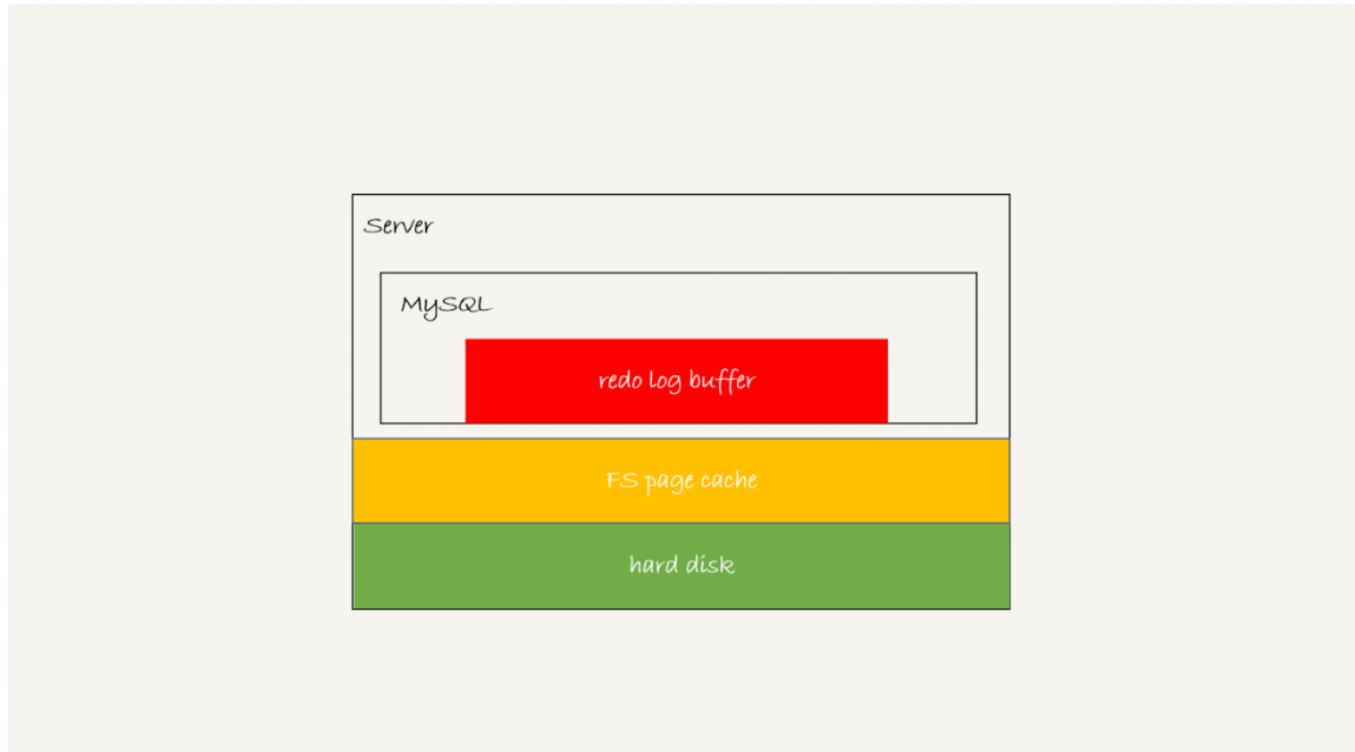


图2 MySQL redo log存储状态

这三种状态分别是：

1. 存在redo log buffer中，物理上是在MySQL进程内存中，就是图中的红色部分；
2. 写到磁盘(write)，但是没有持久化(fsync)，物理上是在文件系统的page cache里面，也就

是图中的黄色部分；

3. 持久化到磁盘，对应的是hard disk，也就是图中的绿色部分。

日志写到redo log buffer是很快的，write到page cache也差不多，但是持久化到磁盘的速度就慢多了。

为了控制redo log的写入策略，InnoDB提供了innodb_flush_log_at_trx_commit参数，它有三种可能取值：

1. 设置为0的时候，表示每次事务提交时都只是把redo log留在redo log buffer中；
2. 设置为1的时候，表示每次事务提交时都将redo log直接持久化到磁盘；
3. 设置为2的时候，表示每次事务提交时都只是把redo log写到page cache。

InnoDB有一个后台线程，每隔1秒，就会把redo log buffer中的日志，调用write写到文件系统的page cache，然后调用fsync持久化到磁盘。

注意，事务执行中间过程的redo log也是直接写在redo log buffer中的，这些redo log也会被后台线程一起持久化到磁盘。也就是说，一个没有提交的事务的redo log，也是可能已经持久化到磁盘的。

实际上，除了后台线程每秒一次的轮询操作外，还有两种场景会让一个没有提交的事务的redo log写入到磁盘中。

1. 一种是，redo log buffer占用的空间即将达到 innodb_log_buffer_size一半的时候，后台线程会主动写盘。注意，由于这个事务并没有提交，所以这个写盘动作只是write，而没有调用fsync，也就是只留在了文件系统的page cache。
2. 另一种是，并行的事务提交的时候，顺带将这个事务的redo log buffer持久化到磁盘。假设一个事务A执行到一半，已经写了一些redo log到buffer中，这时候有另外一个线程的事务B提交，如果innodb_flush_log_at_trx_commit设置的是1，那么按照这个参数的逻辑，事务B要把redo log buffer里的日志全部持久化到磁盘。这时候，就会带上事务A在redo log buffer里的日志一起持久化到磁盘。

这里需要说明的是，我们介绍两阶段提交的时候说过，时序上redo log先prepare，再写binlog，最后再把redo log commit。

如果把innodb_flush_log_at_trx_commit设置成1，那么redo log在prepare阶段就要持久化一次，因为有一个崩溃恢复逻辑是要依赖于prepare的redo log，再加上binlog来恢复的。（如果你印象有点儿模糊了，可以再回顾下[第15篇文章](#)中的相关内容）。

每秒一次后台轮询刷盘，再加上崩溃恢复这个逻辑，InnoDB就认为redo log在commit的时候就不需要fsync了，只会write到文件系统的page cache中就够了。

通常我们说MySQL的“双1”配置，指的就是sync_binlog和innodb_flush_log_at_trx_commit都设置成1。也就是说，一个事务完整提交前，需要等待两次刷盘，一次是redo log（prepare阶段），一次是binlog。

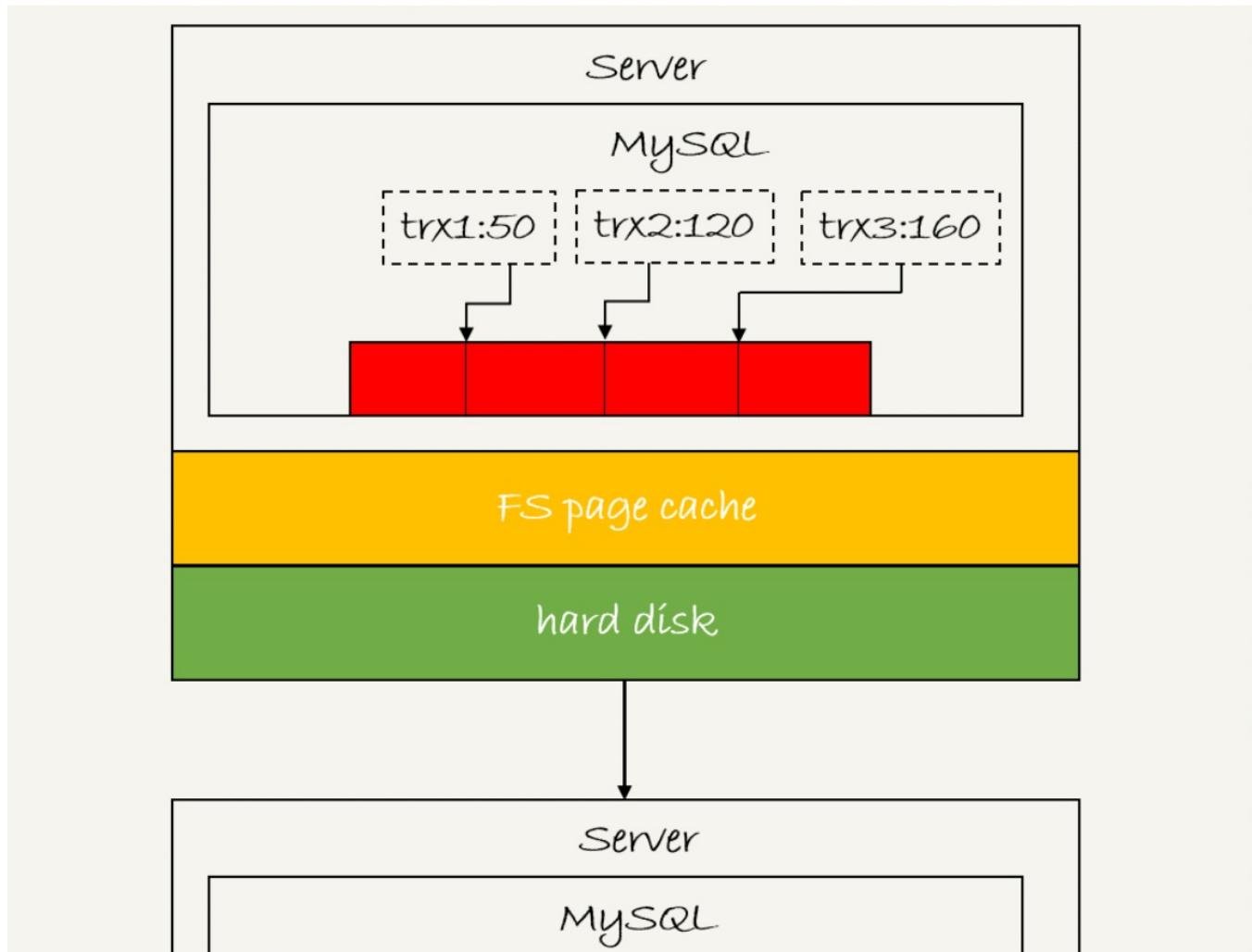
这时候，你可能有一个疑问，这意味着我从MySQL看到的TPS是每秒两万的话，每秒就会写四万次磁盘。但是，我用工具测试出来，磁盘能力也就两万左右，怎么能实现两万的TPS？

解释这个问题，就要用到组提交（group commit）机制了。

这里，我需要先和你介绍日志逻辑序列号（log sequence number, LSN）的概念。LSN是单调递增的，用来对应redo log的一个个写入点。每次写入长度为length的redo log，LSN的值就会加上length。

LSN也会写到InnoDB的数据页中，来确保数据页不会被多次执行重复的redo log。关于LSN和redo log、checkpoint的关系，我会在后面的文章中详细展开。

如图3所示，是三个并发事务(trx1, trx2, trx3)在prepare阶段，都写完redo log buffer，持久化到磁盘的过程，对应的LSN分别是50、120 和160。



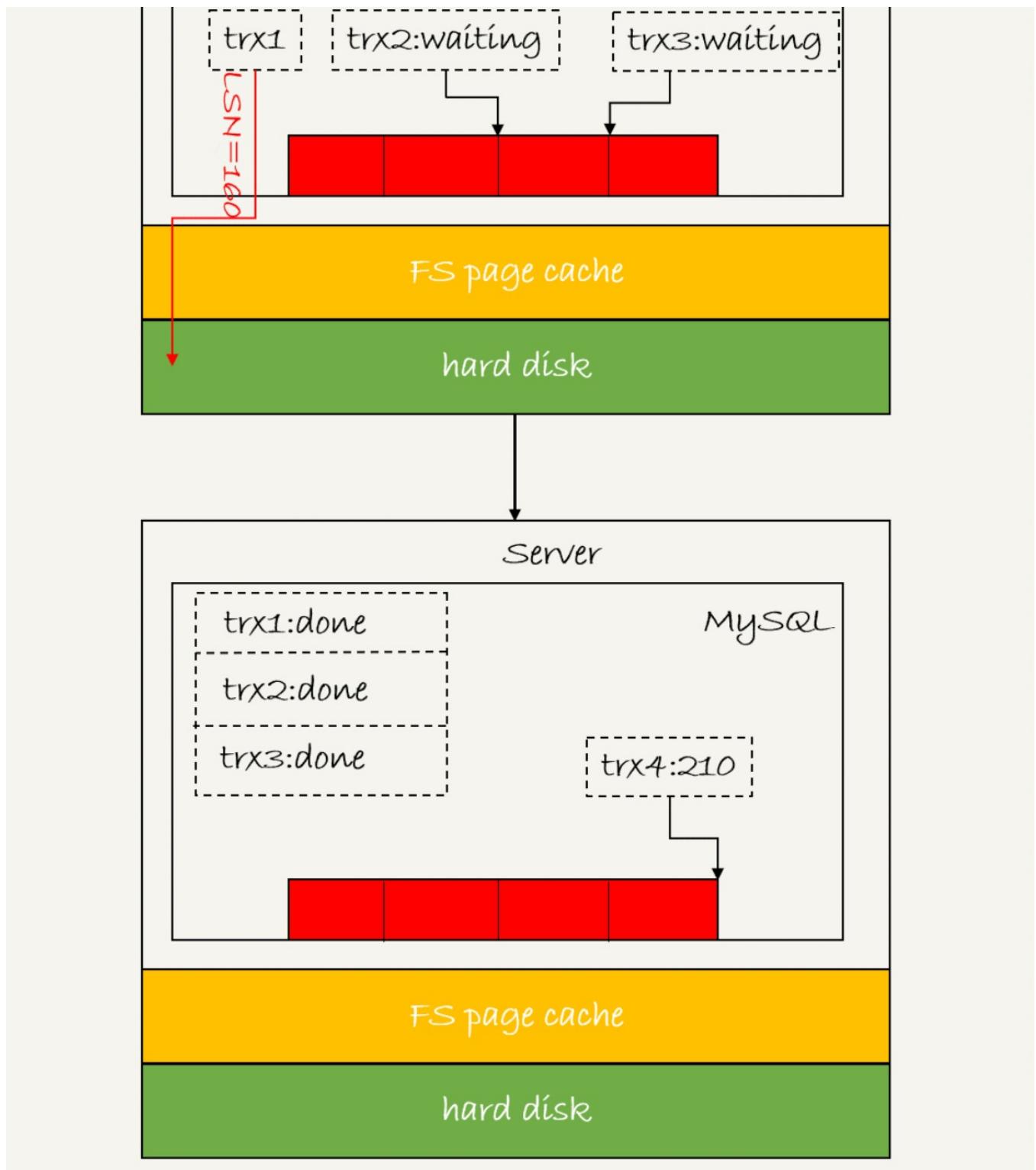


图3 redo log 组提交

从图中可以看到，

1. `trx1`是第一个到达的，会被选为这组的 **leader**；
2. 等`trx1`要开始写盘的时候，这个组里面已经有了三个事务，这时候LSN也变成了160；
3. `trx1`去写盘的时候，带的就是`LSN=160`，因此等`trx1`返回时，所有`LSN`小于等于160的redo log，都已经被持久化到磁盘；

4. 这时候trx2和trx3就可以直接返回了。

所以，一次组提交里面，组员越多，节约磁盘**IOPS**的效果越好。但如果只有单线程压测，那就只能老老实实地一个事务对应一次持久化操作了。

在并发更新场景下，第一个事务写完**redo log buffer**以后，接下来这个**fsync**越晚调用，组员可能越多，节约**IOPS**的效果就越好。

为了让一次**fsync**带的组员更多，**MySQL**有一个很有趣的优化：拖时间。在介绍两阶段提交的时候，我曾经给你画了一个图，现在我把它截过来。

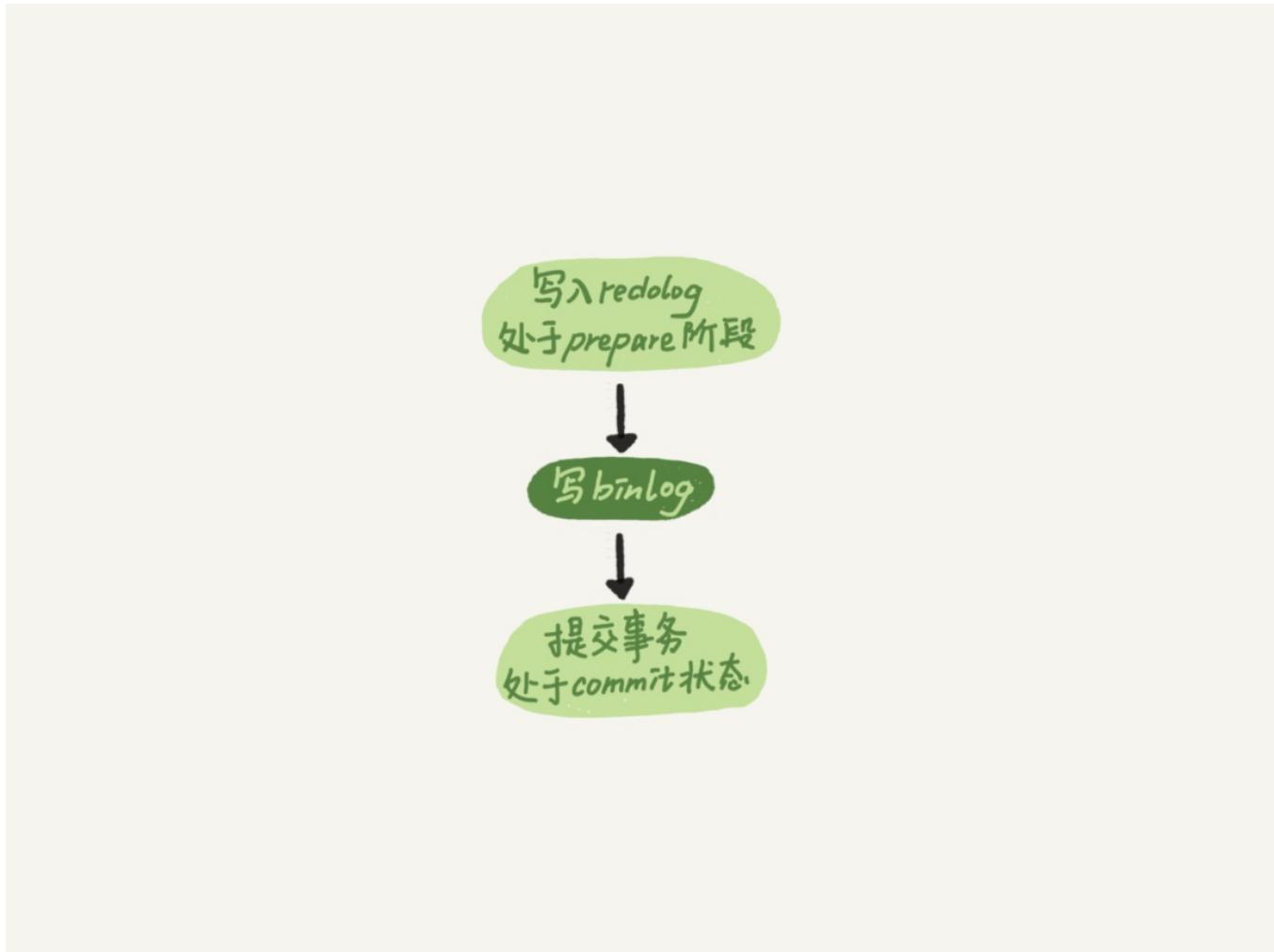


图4 两阶段提交

图中，我把“写**binlog**”当成一个动作。但实际上，写**binlog**是分成两步的：

1. 先把**binlog**从**binlog cache**中写到磁盘上的**binlog**文件；
2. 调用**fsync**持久化。

MySQL为了让组提交的效果更好，把**redo log**做**fsync**的时间拖到了步骤1之后。也就是说，上面的图变成了这样：

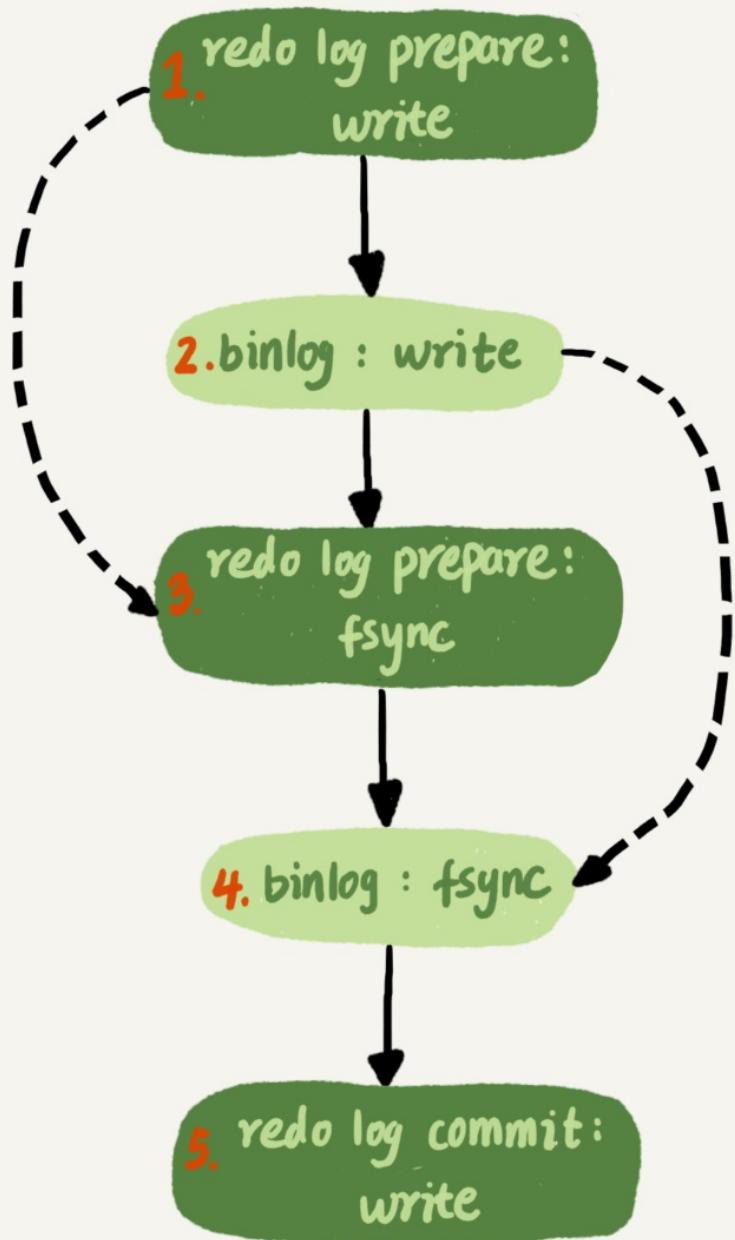


图5 两阶段提交细化

这么一来，**binlog**也可以组提交了。在执行图5中第4步把**binlog fsync**到磁盘时，如果有多个事务的**binlog**已经写完了，也是一起持久化的，这样也可以减少**IOPS**的消耗。

不过通常情况下第3步执行得会很快，所以binlog的write和fsync间的间隔时间短，导致能集合到一起持久化的binlog比较少，因此binlog的组提交的效果通常不如redo log的效果那么好。

如果你想提升binlog组提交的效果，可以通过设置 binlog_group_commit_sync_delay 和 binlog_group_commit_sync_no_delay_count 来实现。

1. binlog_group_commit_sync_delay参数，表示延迟多少微妙后才调用fsync；
2. binlog_group_commit_sync_no_delay_count参数，表示累积多少次以后才调用fsync。

这两个条件是或的关系，也就是说只要有一个满足条件就会调用fsync。

所以，当binlog_group_commit_sync_delay设置为0的时候，binlog_group_commit_sync_no_delay_count也无效了。

之前有同学在评论区问到，WAL机制是减少磁盘写，可是每次提交事务都要写redo log和binlog，这磁盘读写次数也没变少呀？

现在你就能理解了，WAL机制主要得益于两个方面：

1. redo log 和 binlog都是顺序写，磁盘的顺序写比随机写速度要快；
2. 组提交机制，可以大幅度降低磁盘的IOPS消耗。

分析到这里，我们再来回答这个问题：如果你的MySQL现在出现了性能瓶颈，而且瓶颈在IO上，可以通过哪些方法来提升性能呢？

针对这个问题，可以考虑以下三种方法：

1. 设置 binlog_group_commit_sync_delay 和 binlog_group_commit_sync_no_delay_count 参数，减少binlog的写盘次数。这个方法是基于“额外的故意等待”来实现的，因此可能会增加语句的响应时间，但没有丢失数据的风险。
2. 将sync_binlog 设置为大于1的值（比较常见是100~1000）。这样做的风险是，主机掉电时会丢binlog日志。
3. 将innodb_flush_log_at_trx_commit设置为2。这样做的风险是，主机掉电的时候会丢数据。

我不建议你把innodb_flush_log_at_trx_commit 设置成0。因为把这个参数设置成0，表示redo log只保存在内存中，这样的话MySQL本身异常重启也会丢数据，风险太大。而redo log写到文件系统的page cache的速度也是很快的，所以将这个参数设置成2跟设置成0其实性能差不多，但这样做MySQL异常重启时就不会丢数据了，相比之下风险会更小。

小结

在专栏的[第2篇](#)和[第15篇](#)文章中，我和你分析了，如果redo log和binlog是完整的，MySQL是如何保证crash-safe的。今天这篇文章，我着重和你介绍的是MySQL是“怎么保证redo log和binlog是完整的”。

希望这三篇文章串起来的内容，能够让你对crash-safe这个概念有更清晰的理解。

之前的第15篇答疑文章发布之后，有同学继续留言问到了一些跟日志相关的问题，这里为了方便你回顾、学习，我再集中回答一次这些问题。

问题1：执行一个update语句以后，我再去执行hexdump命令直接查看ibd文件内容，为什么没有看到数据有改变呢？

回答：这可能是因为WAL机制的原因。update语句执行完成后，InnoDB只保证写完了redo log、内存，可能还没来得及将数据写到磁盘。

问题2：为什么binlog cache是每个线程自己维护的，而redo log buffer是全局共用的？

回答：MySQL这么设计的主要原因是，binlog是不能“被打断的”。一个事务的binlog必须连续写，因此要整个事务完成后，再一起写到文件里。

而redo log并没有这个要求，中间有生成的日志可以写到redo log buffer中。redo log buffer中的内容还能“搭便车”，其他事务提交的时候可以被一起写到磁盘中。

问题3：事务执行期间，还没到提交阶段，如果发生crash的话，redo log肯定丢了，这会不会导致主备不一致呢？

回答：不会。因为这时候binlog也还在binlog cache里，没发给备库。crash以后redo log和binlog都没有了，从业务角度看这个事务也没有提交，所以数据是一致的。

问题4：如果binlog写完盘以后发生crash，这时候还没给客户端答复就重启了。等客户端再重连进来，发现事务已经提交成功了，这是不是bug？

回答：不是。

你可以设想一下更极端的情况，整个事务都提交成功了，redo log commit完成了，备库也收到binlog并执行了。但是主库和客户端网络断开了，导致事务成功的包返回不回去，这时候客户端也会收到“网络断开”的异常。这种也只能算是事务成功的，不能认为是bug。

实际上数据库的crash-safe保证的是：

1. 如果客户端收到事务成功的消息，事务就一定持久化了；
2. 如果客户端收到事务失败（比如主键冲突、回滚等）的消息，事务就一定失败了；

3. 如果客户端收到“执行异常”的消息，应用需要重连后通过查询当前状态来继续后续的逻辑。

此时数据库只需要保证内部（数据和日志之间，主库和备库之间）一致就可以了。

最后，又到了课后问题时间。

今天我留给你的思考题是：你的生产库设置的是“双1”吗？如果平时是的话，你有在什么场景下改成过“非双1”吗？你的这个操作又是基于什么决定的？

另外，我们都知道这些设置可能有损，如果发生了异常，你的止损方案是什么？

你可以把你的理解或者经验写在留言区，我会在下一篇文章的末尾选取有趣的评论和你一起分享和分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上篇文章最后，想要你分享的是线上“救火”的经验。

@Long 同学，在留言中提到了几个很好的场景。

- 其中第3个问题，“如果一个数据库是被客户端的压力打满导致无法响应的，重启数据库是没有用的。”，说明他很好地思考了。

这个问题是因为重启之后，业务请求还会再发。而且由于是重启，**buffer pool**被清空，可能会导致语句执行得更慢。

- 他提到的第4个问题也很典型。有时候一个表上会出现多个单字段索引（而且往往这是因为运维工程师对索引原理不够清晰做的设计），这样就可能出现优化器选择索引合并算法的现象。但实际上，索引合并算法的效率并不好。而通过将其中的一个索引改成联合索引的方法，是一个很好的应对方案。

还有其他几个同学提到的问题场景，也很好，很值得你一看。

@Max 同学提到一个很好的例子：客户端程序的连接器，连接完成后会做一些诸如**show columns**的操作，在短连接模式下这个影响就非常大了。

这个提醒我们，在**review**项目的时候，不止要**review**我们自己业务的代码，也要**review**连接器的行为。一般做法就是在测试环境，把**general_log**打开，用业务行为触发连接，然后通过**general log**分析连接器的行为。

@Manjusaka 同学的留言中，第二点提得非常好：如果你的数据库请求模式直接对应于客户请求，这往往是一个危险的设计。因为客户行为不可控，可能突然因为你们公司的一个运营推广，压力暴增，这样很容易把数据库打挂。

在设计模型里面设计一层，专门负责管理请求和数据库服务资源，对于比较重要和大流量的业务，是一个好的设计方向。

@Vincent 同学提了一个好问题，用文中提到的**DDL**方案，会导致**binlog**里面少了这个**DDL**语句，后续影响备份恢复的功能。由于需要另一个知识点（主备同步协议），我放在后面的文章中说明。



The image shows the cover of a MySQL course. At the top left is the '极客时间' logo. The title 'MySQL 实战 45 讲' is prominently displayed in large, bold, dark font. Below it is the subtitle '从原理到实战，丁奇带你搞懂 MySQL' in a smaller, lighter font. On the right side of the cover, there is a portrait photo of the instructor, Ding Qi, wearing glasses and a black shirt, with his arms crossed. At the bottom left, the author's name '林晓斌' and '网名丁奇 前阿里资深技术专家' are listed. A promotional message at the bottom right encourages users to upgrade to the new version and invite friends to read for free.

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

精选留言



锅子

2

老师好，有一个疑问：当设置**sync_binlog=0**时，每次**commit**都只时**write**到**page cache**，并不会**fsync**。但是做实验时**binlog**文件中还是会有记录，这是什么原因呢？是不是后台线程每秒一次的轮询也会将**binlog cache**持久化到磁盘？还是有其他的参数控制呢？

2019-01-04

| 作者回复

你看到的“**binlog**的记录”，也是从**page cache**读的哦。

Page cache是操作系统文件系统上的

好问题

2019-01-04



倪大人

4

老师求解**sync_binlog**和**binlog_group_commit_sync_no_delay_count**这两个参数区别

如果

`sync_binlog = N`
`binlog_group_commit_sync_no_delay_count = M`
`binlog_group_commit_sync_delay = 很大值`
这种情况`fsync`什么时候发生呀，`min(N,M)`吗？
感觉`sync_binlog`搭配`binlog_group_commit_sync_delay`也可以实现组提交？

如果

`sync_binlog = 0`
`binlog_group_commit_sync_no_delay_count = 10`
这种情况下是累计10个事务`fsync`一次？

2019-01-04

作者回复

好问题，我写这篇文章的时候也为了这个问题去翻了代码，是这样的：

达到N次以后，可以刷盘了，然后再进入(`sync_delay`和`no_delay_count`)这个逻辑；

`Sync_delay`如果很大，就达到`no_delay_count`才刷；

只要`sync_binlog=0`，也会有前面的等待逻辑，但是等完后还是不调`fsync`！

2019-01-06



WilliamX

3

为什么`binlog cache`是每个线程自己维护的，而`redo log buffer`是全局共用的？

这个问题，感觉还有一点，`binlog`存储是以`statement`或者`row`格式存储的，而`redo log`是以`page`页格式存储的。`page`格式，天生就是共有的，而`row`格式，只跟当前事务相关

2019-01-04

作者回复

嗯，这个解释也很好。回

2019-01-04



一大只

2

你是怎么验证的？等于0的时候虽然有走这个逻辑，但是最后调用`fsync`之前判断是0，就啥也没做就走了

回复老师：

老师，我说的`sync_binlog=0`或`=1`效果一样，就是看语句实际执行的效果，参数`binlog_group_commit_sync_delay`我设置成了500000微秒，在`=1`或`=0`时，对表进行`Insert`，然后都会有0.5秒的等待，也就是执行时间都是0.51 sec，关闭`binlog_group_commit_sync_delay`，`insert`执行会飞快，所以我认为`=1`或`=0`都是受组提交参数的影响的。

2019-01-05

作者回复

回

非常好

然后再补上我回答的这个逻辑，就完备了

2019-01-05

alias cd=rm -rf

1

事务A是当前事务，这时候事务B提交了。事务B的redolog持久化时候，会顺道把A产生的redolog也持久化，这时候A的redolog状态是prepare状态么？

2019-01-28

| 作者回复

不是。

说明一下哈，所谓的 redo log prepare，是“当前事务提交”的一个阶段，也就是说，在事务A提交的时候，我们才会走到事务A的redo log prepare这个阶段。

事务A在提交前，有一部分redo log被事务B提前持久化，但是事务A还没有进入提交阶段，是无所谓“redo log prepare”的。

好问题

2019-01-28



某、人

1

有调到非双1的时候，在大促时非核心库和从库延迟较多的情况。

设置的是sync_binlog=0和innodb_flush_log_at_trx_commit=2

针对0和2，在mysql crash时不会出现异常，在主机挂了时，会有几种风险：

1. 如果事务的binlog和redo log都还未fsync，则该事务数据丢失

2. 如果事务binlog fsync成功，redo log未fsync，则该事务数据丢失。

虽然binlog落盘成功，但是binlog没有恢复redo log的能力，所以redo log不能恢复。

不过后续可以解析binlog来恢复这部分数据

3. 如果事务binlog fsync未成功，redo log成功。

由于redo log恢复数据是在引擎层，所以重新启动数据库，redo log能恢复数据，但是不能恢复server层的binlog，则binlog丢失。

如果该事务还未从FS page cache里发送给从库，那么主从就会出现不一致的情况

4. 如果binlog和redo log都成功fsync，那么皆大欢喜。

老师我有几个问题：

1. 因为binlog不能被打断，那么binlog做fsync是单线程吧？

如果是的话，那么binlog的write到fsync的时间，就应该是redo log fsync+上一个事务的binlog fsync时间。

但是测试到的现象，一个超大事务做fsync时，对其它事务的提交影响也不大。

如果是多线程做fsync，怎么保证的一个事务binlog在磁盘上的连续性？

2. 5.7的并行复制是基于binlog组成员并行的，为什么很多文章说是表级别的并行复制？

2019-01-06

作者回复

1. Write的时候只要写进去了，`fsync`其实很快的。连续性是`write`的时候做的（写的时候保证了连续）

2. 你的理解应该是对的。不是表级

2019-01-06



永恒记忆

1

主从模式下，内网从库如果设置双1，刚还原的数据发现根本追不上主库，所以从库设置了0，老师后面章节会讲关于mysql包括主从监控这块的内容吗。

2019-01-04

作者回复

会讲到

2019-01-04



往事随风，顺其自然

1

`redolog` 里面有已经提交事物日志，还有未提交事物日志都持久化到磁盘，此时异常重启，`binlog` 里面不是多余记录的未提交事物，干嘛不设计不添加未提交事物不更好

2019-01-04



miu

0

老师，关于`BINLOG_GROUP_COMMIT_SYNC_DELAY`,
`BINLOG_GROUP_COMMIT_SYNC_NO_DELAY_COUNT`,
`SYNC_BINLOG`三个参数，我的理解是：

若`SYNC_BINLOG>1`时，且设置了`BINLOG_GROUP_COMMIT_SYNC_DELAY`和`BINLOG_GROUP_COMMIT_SYNC_NO_DELAY_COUNT`两个参数。

例如

`sync_binlog=2,`
`BINLOG_GROUP_COMMIT_SYNC_DELAY=1000000,`
`BINLOG_GROUP_COMMIT_SYNC_NO_DELAY_COUNT=3,`

那么在执行完第1个事务后，在第2个事务提交时，会根据后续的事务提交来判断`fsync`等待的时间，

若后续在1秒内没有累积3个事务的提交，则会等待1秒后再做`fsync`，从SQL语句来看，执行第一个语句很快，第二个语句需要等待1秒才成功。这时延时等待的时间是`BINLOG_GROUP_COMMIT_SYNC_DELAY`所设置的值。

若执行完第1个事务后，并行执行3个事务（1秒内完成），则后续3个事务会同时做`fsync`，这时延时等待的时间是`BINLOG_GROUP_COMMIT_SYNC_NO_DELAY_COUNT`设置的数量的事务提交的间隔时间。

也就是`sync_binlog+BINLOG_GROUP_COMMIT_SYNC_NO_DELAY_COUNT-1`个事务做一次`fsync`。

我测试的版本是MySQL官方5.7.24，请老师点评。

2019-02-01

作者回复

这两个逻辑不建议放到一起算

就是按照这样：

1. 有设置 `BINLOG_GROUP_COMMIT_SYNC_NO_DELAY_COUNT` 这个值，（假设 `SYNC_DELAY` 很大），提交的时候就得等这么多次才能过；
2. 到了提交阶段，又要按照 `sync_binlog` 来判断是否刷盘。

新春快乐~

2019-02-04

0

alias cd=rm -rf

老师不好意思，我接着刚才的问题问哈

并发事务的 `redolog` 持久化，会把当前事务的 `redolog` 持久化，当前事务的 `redolog` 持久化后 `prepare` 状态么？`redolog` 已经被持久化到磁盘了，那么当前事务提交时候，`redolog` 变为 `prepare` 状态，这时候是从 `redologbuffer` 加载还是从磁盘加载？

2019-01-28

作者回复

每个事务在提交过程的 `prepare` 阶段，会把 `redolog` 持久化；“当前事务的 `redolog` 持久化后 `prepare` 状态么”这个描述还是不清楚，你用事务 A、事务 B 这样来描述吧！

`redolog` 已经被持久化到磁盘了，那么当前事务提交时候，

（其实这里只是“部分”被持久化，因为这个事务自己在执行的过程中，还会产生新的日志），只需要继续持久化剩下的 `redo log`

2019-01-28

0

alias cd=rm -rf

您好，我看文章后有俩点疑问，前提条件如果 `mysql` 设置双 1

1. 这时候磁盘中的 `redolog` 的状态是什么状态呢？是 `prepare` 么？
2. 如果一个事务在进行中的时候 `redolog` 已经被持久化，在事务提交时候，这条 `redolog` 还在 `redolog-buffer` 中么？

2019-01-27

作者回复

1. “这时候磁盘中的 `redolog` 的状态是什么状态呢？是 `prepare` 么？”这个“这时候”是什么意思？
2. 还在，不过随时可以被覆盖

2019-01-28

0

嘻嘻

1. 如果客户端收到事务成功的消息，事务就一定持久化了；
`commit` 是在什么阶段返回的？如果写完 `page cache` 就返回也没有持久化吧？

2019-01-25

作者回复

第一个问题没看懂。

“如果写完page cache就返回也没有持久化吧”， 是的，

“客户端收到事务成功的消息， 事务就一定持久化了”是建立在双1基础上的。

2019-01-26



Geek_527020

0

您好，老师，我有一个以后，组提交，把为提交事务的redo log写入磁盘，如果有查询，岂不是查到未提交事务的更新内容了？

2019-01-25

作者回复

不会啊，有MVCC的， 08篇再看下

2019-01-25



JI

0

共同写一个binlog文件，这个过程应该需要锁来维持提交的时序吧，写文件的时候是不是可能会变成瓶颈点？

2019-01-23

作者回复

不会的，大家分头写，然后一起持久化到磁盘

2019-01-23



Komine

0

为什么binlog 是不能“被打断的”的呢？主要出于什么考虑？

2019-01-22

作者回复

好问题

我觉得一个比较重要的原因是，一个线程只能同时有一个事务在执行。

由于这个设定，所以每当执行一个begin/start transaction的时候，就会默认提交上一个事务；这样如果一个事务的binlog被拆开的时候，在备库执行就会被当做多个事务分段执行，这样破坏了原子性，是有问题的。

2019-01-22



就是个渣渣

0

林老师，你好！超过了binlog_cache_size，暂存到磁盘，那如果超过了max_binlog_cache_size就直接报错了呢，这两个参数的关联是什么呢？

2019-01-19

作者回复

`max_binlog_cache_size`只是用来限制设置`binlog_cache_size`的时候的上限

并不参与执行语句的逻辑的

2019-01-19



似水流年

0

我网上查`pagecache`是在内存里的，这与您讲的一样吗？

2019-01-15

| 作者回复

就是文件系统的`page cache`，是属于操作系统的内存的一部分

2019-01-15



猪哥哥

0

老师好，能说下`innodb_log_buffer_size`参数的作用吗

2019-01-10



roaming

0

看了几遍，终于看明白了

2019-01-10

| 作者回复

0

2019-01-10



猪哥哥

0

老师 我想问下文件系统的`page cache`还是不是内存，是不是文件系统向内核申请的一块的内存？

2019-01-10

| 作者回复

你理解的是对的

2019-01-10

24 | MySQL是怎么保证主备一致的？

2019-01-07 林晓斌



在前面的文章中，我不止一次地和你提到了**binlog**，大家知道**binlog**可以用来归档，也可以用来做主备同步，但它的内容是什么样的呢？为什么备库执行了**binlog**就可以跟主库保持一致了呢？今天我就正式地和你介绍一下它。

毫不夸张地说，MySQL能够成为现下最流行的开源数据库，**binlog**功不可没。

在最开始，MySQL是以容易学习和方便的高可用架构，被开发人员青睐的。而它的几乎所有的高可用架构，都直接依赖于**binlog**。虽然这些高可用架构已经呈现出越来越复杂的趋势，但都是从最基本的一主一备演化过来的。

今天这篇文章我主要为你介绍主备的基本原理。理解了背后的设计原理，你也可以从业务开发的角度，来借鉴这些设计思想。

MySQL主备的基本原理

如图1所示就是基本的主备切换流程。

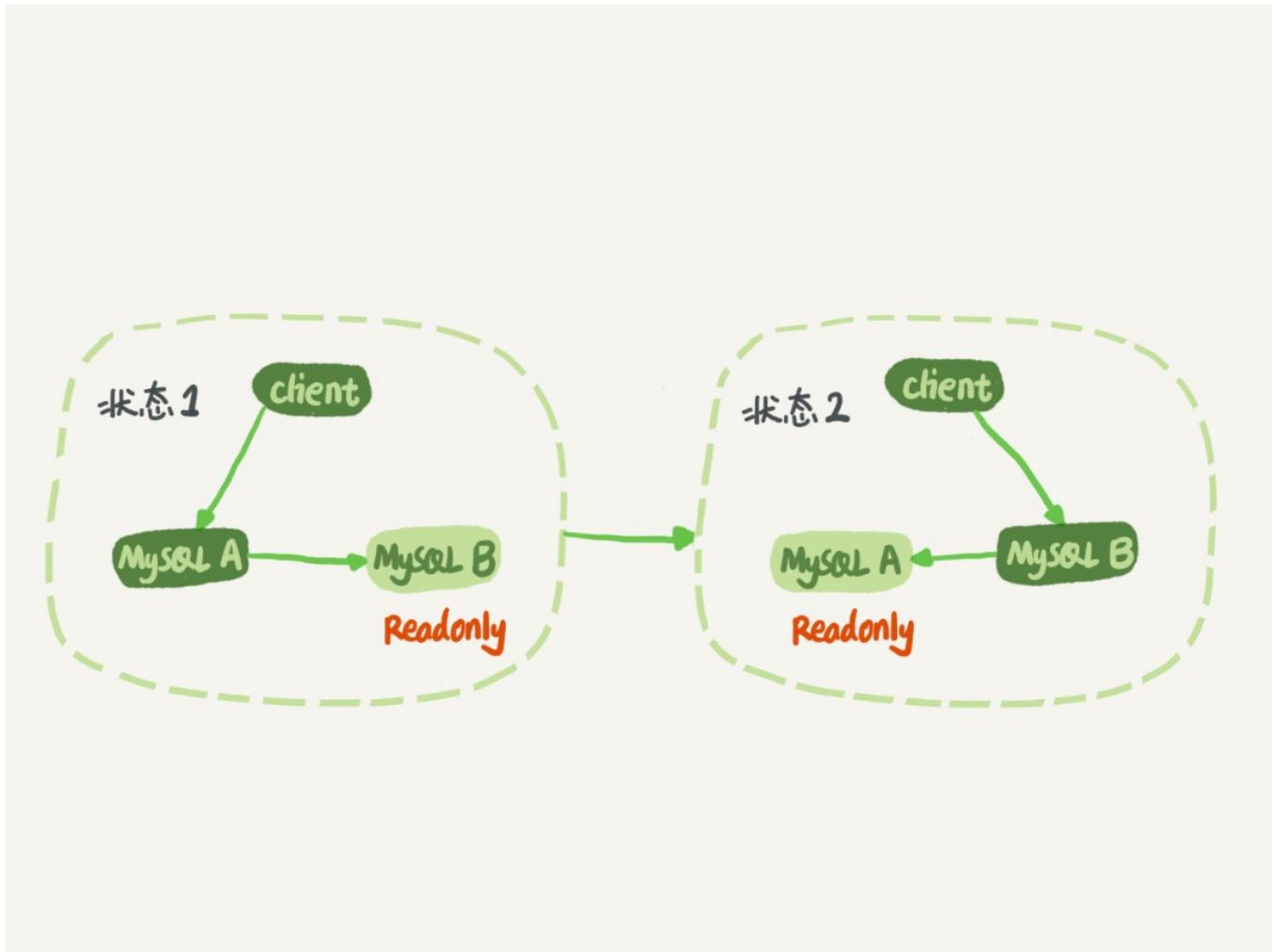


图 1 MySQL主备切换流程

在状态1中，客户端的读写都直接访问节点A，而节点B是A的备库，只是将A的更新都同步过来，到本地执行。这样可以保持节点B和A的数据是相同的。

当需要切换的时候，就切成状态2。这时候客户端读写访问的都是节点B，而节点A是B的备库。

在状态1中，虽然节点B没有被直接访问，但是我依然建议你把节点B（也就是备库）设置成只读（readonly）模式。这样做，有以下几个考虑：

1. 有时候一些运营类的查询语句会被放到备库上去查，设置为只读可以防止误操作；
2. 防止切换逻辑有bug，比如切换过程中出现双写，造成主备不一致；
3. 可以用readonly状态，来判断节点的角色。

你可能会问，我把备库设置成只读了，还怎么跟主库保持同步更新呢？

这个问题，你不用担心。因为readonly设置对超级(super)权限用户是无效的，而用于同步更新的线程，就拥有超级权限。

接下来，我们再看看节点A到B这条线的内部流程是什么样的。图2中画出的就是一个update

语句在节点A执行，然后同步到节点B的完整流程图。

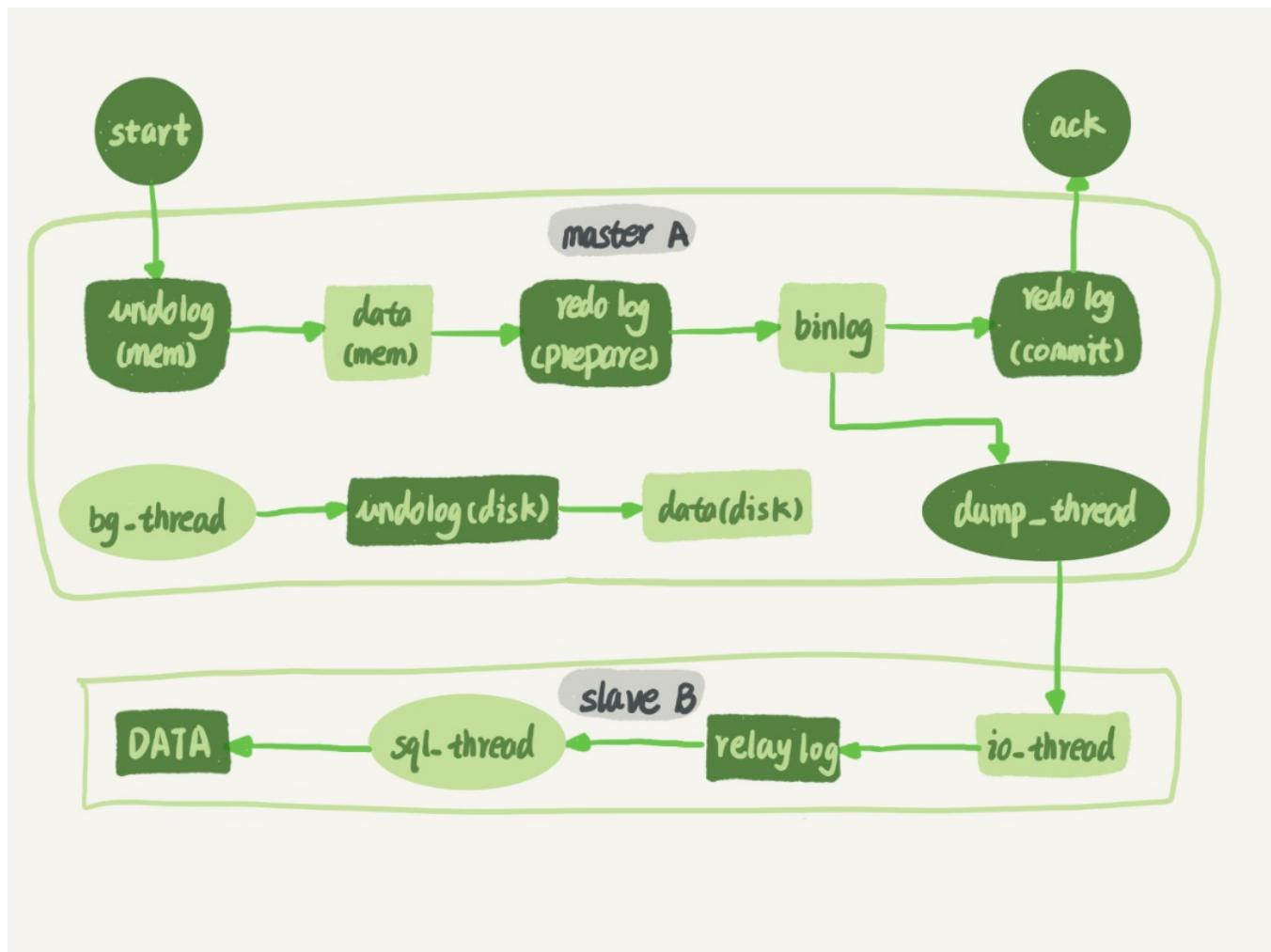


图2 主备流程图

图2中，包含了我在上一篇文章中讲到的binlog和redo log的写入机制相关的内容，可以看到：主库接收到客户端的更新请求后，执行内部事务的更新逻辑，同时写binlog。

备库B跟主库A之间维持了一个长连接。主库A内部有一个线程，专门用于服务备库B的这个长连接。一个事务日志同步的完整过程是这样的：

1. 在备库B上通过**change master**命令，设置主库A的IP、端口、用户名、密码，以及要从哪个位置开始请求binlog，这个位置包含文件名和日志偏移量。
2. 在备库B上执行**start slave**命令，这时候备库会启动两个线程，就是图中的**io_thread**和**sql_thread**。其中**io_thread**负责与主库建立连接。
3. 主库A校验完用户名、密码后，开始按照备库B传过来的位置，从本地读取binlog，发给B。
4. 备库B拿到binlog后，写到本地文件，称为中转日志（**relay log**）。
5. **sql_thread**读取中转日志，解析出日志里的命令，并执行。

这里需要说明，后来由于多线程复制方案的引入，`sql_thread`演化成为了多个线程，跟我们今天要介绍的原理没有直接关系，暂且不展开。

分析完了这个长连接的逻辑，我们再来看一个问题：`binlog`里面到底是什么内容，为什么备库拿过去可以直接执行。

binlog的三种格式对比

我在[第15篇答疑文章](#)中，和你提到过`binlog`有两种格式，一种是`statement`，一种是`row`。可能你在其他资料上还会看到有第三种格式，叫作`mixed`，其实它就是前两种格式的混合。

为了便于描述`binlog`的这三种格式间的区别，我创建了一个表，并初始化几行数据。

```
mysql> CREATE TABLE `t` (
    `id` int(11) NOT NULL,
    `a` int(11) DEFAULT NULL,
    `t_modified` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    KEY `a` (`a`),
    KEY `t_modified`(`t_modified`)
) ENGINE=InnoDB;
```

```
insert into t values(1,1,'2018-11-13');
insert into t values(2,2,'2018-11-12');
insert into t values(3,3,'2018-11-11');
insert into t values(4,4,'2018-11-10');
insert into t values(5,5,'2018-11-09');
```

如果要在表中删除一行数据的话，我们来看看这个`delete`语句的`binlog`是怎么记录的。

注意，下面这个语句包含注释，如果你用MySQL客户端来做这个实验的话，要记得加`-c`参数，否则客户端会自动去掉注释。

```
mysql> delete from t /*comment*/ where a>=4 and t_modified<='2018-11-10' limit 1;
```

当`binlog_format=statement`时，`binlog`里面记录的就是SQL语句的原文。你可以用

```
mysql> show binlog events in 'master.000001';
```

命令看binlog中的内容。

```
| master.000001 | 5889 | Anonymous_Gtid |      1 |      5954 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'
| master.000001 | 5954 | Query       |      1 |      6041 | BEGIN
| master.000001 | 6041 | Query       |      1 |      6197 | use `test`; delete from t /*comment*/ where a>=4 and t_modified<='2018-11-10' limit 1
| master.000001 | 6197 | Xid        |      1 |      6228 | COMMIT /* xid=61 */
```

图3 statement格式binlog示例

现在，我们来看一下图3的输出结果。

- 第一行`SET @@SESSION.GTID_NEXT='ANONYMOUS'`你可以先忽略，后面文章我们会在介绍主备切换的时候再提到；
- 第二行是一个`BEGIN`，跟第四行的`commit`对应，表示中间是一个事务；
- 第三行就是真实执行的语句了。可以看到，在真实执行的`delete`命令之前，还有一个“`use 'test'`”命令。这条命令不是我们主动执行的，而是MySQL根据当前要操作的表所在的数据库，自行添加的。这样做可以保证日志传到备库去执行的时候，不论当前的工作线程在哪个库里，都能够正确地更新到`test`库的表`t`。

`use 'test'`命令之后的`delete`语句，就是我们输入的SQL原文了。可以看到，binlog“忠实”地记录了SQL命令，甚至连注释也一并记录了。

- 最后一行是一个`COMMIT`。你可以看到里面写着`xid=61`。你还记得这个`XID`是做什么用的吗？如果记忆模糊了，可以再回顾一下[第15篇文章](#)中的相关内容。

为了说明statement 和 row格式的区别，我们来看一下这条`delete`命令的执行效果图：

```
mysql> show warnings;
+-----+-----+
| Level | Code | Message
+-----+-----+
| Note | 1592 | Unsafe statement written to the binary log using statement format since BINLOG_FORMAT = STATEMENT. The statement is unsafe because it uses a LIMIT clause. This is unsafe because the set of rows included cannot be predicted. |
+-----+-----+
1 row in set (0.00 sec)
```

图4 delete执行warnings

可以看到，运行这条`delete`命令产生了一个warning，原因是当前binlog设置的是statement格式，并且语句中有`limit`，所以这个命令可能是unsafe的。

为什么这么说呢？这是因为`delete`带`limit`，很可能会出现主备数据不一致的情况。比如上面这个例子：

- 如果`delete`语句使用的是索引`a`，那么会根据索引`a`找到第一个满足条件的行，也就是说删除的是`a=4`这一行；
- 但如果使用的是索引`t_modified`，那么删除的就是`t_modified='2018-11-09'`也就是`a=5`这一行。

由于**statement**格式下，记录到**binlog**里的是语句原文，因此可能会出现这样一种情况：在主库执行这条**SQL**语句的时候，用的是索引**a**；而在备库执行这条**SQL**语句的时候，却使用了索引**t_modified**。因此，**MySQL**认为这样写是有风险的。

那么，如果我把**binlog**的格式改为**binlog_format='row'**，是不是就没有这个问题了呢？我们先来看看这时候**binlog**中的内容吧。

master.000001 8900 Anonymous_Gtid	1	8965 SET @@SESSION.GTID_NEXT= 'ANONYMOUS'
master.000001 8965 Query	1	9045 BEGIN
master.000001 9045 Table_map	1	9092 table_id: 226 (test.t)
master.000001 9092 Delete_rows	1	9140 table_id: 226 flags: STMT_END_F
master.000001 9140 Xid	1	9171 COMMIT /* xid=68 */

图5 row格式**binlog**示例

可以看到，与**statement**格式的**binlog**相比，前后的**BEGIN**和**COMMIT**是一样的。但是，**row**格式的**binlog**里没有了**SQL**语句的原文，而是替换成了两个**event**: **Table_map**和**Delete_rows**。

1. **Table_map event**, 用于说明接下来要操作的表是**test**库的表**t**;
2. **Delete_rows event**, 用于定义删除的行为。

其实，我们通过图5是看不到详细信息的，还需要借助**mysqlbinlog**工具，用下面这个命令解析和查看**binlog**中的内容。因为图5中的信息显示，这个事务的**binlog**是从**8900**这个位置开始的，所以可以用**start-position**参数来指定从这个位置的日志开始解析。

```
mysqlbinlog -w data/master.000001 --start-position=8900;
```

```
BEGIN
/*!*/;
# at 9045
#181229 23:32:22 server id 1  end_log_pos 9092 CRC32 0xdbfc0a8c      Table_map: `test`.`t` mapped to number 226
# at 9092
#181229 23:32:22 server id 1  end_log_pos 9140 CRC32 0x0cda8921      Delete_rows: table id 226 flags: STMT_END_F

BINLOG '
hpMnXBMBAAAALwAAAIQjAAAAAOIAAAAAAAEABHRlc3QAXQAAwMDEQEAAowK/Ns=
hpMnXCABAAAAMAAAALQjAAAAAOIAAAAAAAEAgAD//gEAAAABAAAFAv19VAhidoM
'/*!*/;
### DELETE FROM `test`.`t`
### WHERE
###   @1=4 /* INT meta=0 nullable=0 is_null=0 */
###   @2=4 /* INT meta=0 nullable=1 is_null=0 */
###   @3=1541797200 /* TIMESTAMP(0) meta=0 nullable=0 is_null=0 */
# at 9140
#181229 23:32:22 server id 1  end_log_pos 9171 CRC32 0x1beb44f1          Xid = 68
COMMIT/*!*/;
SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlbinlog */ /*!*/;
```

图6 row格式binlog示例的详细信息

从这个图中，我们可以看到以下几个信息：

- **server id 1**，表示这个事务是在**server_id=1**的这个库上执行的。
- 每个**event**都有**CRC32**的值，这是因为我把参数**binlog_checksum**设置成了**CRC32**。
- **Table_map event**跟在图5中看到的相同，显示了接下来要打开的表，**map**到数字**226**。现在我们这条**SQL**语句只操作了一张表，如果要操作多张表呢？每个表都有一个对应的**Table_map event**、都会**map**到一个单独的数字，用于区分对不同表的操作。
- 我们在**mysqlbinlog**的命令中，使用了**-w**参数是为了把内容都解析出来，所以从结果里面可以看到各个字段的值（比如，**@1=4**、**@2=4**这些值）。
- **binlog_row_image**的默认配置是**FULL**，因此**Delete_event**里面，包含了删掉的行的所有字段的值。如果把**binlog_row_image**设置为**MINIMAL**，则只会记录必要的信息，在这个例子里，就是只会记录**id=4**这个信息。
- 最后的**Xid event**，用于表示事务被正确地提交了。

你可以看到，当**binlog_format**使用**row**格式的时候，**binlog**里面记录了真实删除行的主键**id**，这样**binlog**传到备库去的时候，就肯定会删除**id=4**的行，不会有主备删除不同行的问题。

为什么会有**mixed**格式的**binlog**？

基于上面的信息，我们来讨论一个问题：为什么会有**mixed**这种**binlog**格式的存在场景？推论过程是这样的：

- 因为有些**statement**格式的**binlog**可能会导致主备不一致，所以要使用**row**格式。
- 但**row**格式的缺点是，很占空间。比如你用一个**delete**语句删掉10万行数据，用**statement**的话就是一个**SQL**语句被记录到**binlog**中，占用几十个字节的空间。但如果用**row**格式的**binlog**，就要把这10万条记录都写到**binlog**中。这样做，不仅会占用更大的空间，同时写**binlog**也要耗费**IO**资源，影响执行速度。
- 所以，**MySQL**就取了个折中方案，也就是有了**mixed**格式的**binlog**。**mixed**格式的意思是，**MySQL**自己会判断这条**SQL**语句是否可能引起主备不一致，如果有可能，就用**row**格式，否则就用**statement**格式。

也就是说，**mixed**格式可以利用**statement**格式的优点，同时又避免了数据不一致的风险。

因此，如果你的线上**MySQL**设置的**binlog**格式是**statement**的话，那基本上就可以认为这是一个不合理的设置。你至少应该把**binlog**的格式设置为**mixed**。

比如我们这个例子，设置为**mixed**后，就会记录为**row**格式；而如果执行的语句去掉**limit 1**，就会记录为**statement**格式。

当然我要说的是，现在越来越多的场景要求把**MySQL**的**binlog**格式设置成**row**。这么做的理由有

很多，我来给你举一个可以直接看出来的好处：恢复数据。

接下来，我们就分别从`delete`、`insert`和`update`这三种SQL语句的角度，来看看数据恢复的问题。

通过图6你可以看出来，即使我执行的是`delete`语句，`row`格式的`binlog`也会把被删掉的行的整行信息保存起来。所以，如果你在执行完一条`delete`语句以后，发现删错数据了，可以直接把`binlog`中记录的`delete`语句转成`insert`，把被错删的数据插入回去就可以恢复了。

如果你是执行错了`insert`语句呢？那就更直接了。`row`格式下，`insert`语句的`binlog`里会记录所有的字段信息，这些信息可以用来精确定位刚刚被插入的那一行。这时，你直接把`insert`语句转成`delete`语句，删除掉这被误插入的一行数据就可以了。

如果执行的是`update`语句的话，`binlog`里面会记录修改前整行的数据和修改后的整行数据。所以，如果你误执行了`update`语句的话，只需要把这个`event`前后的两行信息对调一下，再去数据库里面执行，就能恢复这个更新操作了。

其实，由`delete`、`insert`或者`update`语句导致的数据操作错误，需要恢复到操作之前状态的情况，也时有发生。[MariaDB的Flashback](#)工具就是基于上面介绍的原理来回滚数据的。

虽然`mixed`格式的`binlog`现在已经用得不多了，但这里我还是要再借用一下`mixed`格式来说明一个问题，来看一下这条SQL语句：

```
mysql> insert into t values(10,10, now());
```

如果我们把`binlog`格式设置为`mixed`，你觉得MySQL会把它记录为`row`格式还是`statement`格式呢？

先不要着急说结果，我们一起来看一下这条语句执行的效果。

```
| master.000001 | 2738 | Query      |      1 |      2825 | BEGIN
| master.000001 | 2825 | Query      |      1 |      2942 | use `test`; insert into t values(100, 1, now())
| master.000001 | 2942 | Xid       |      1 |      2973 | COMMIT /* xid=41 */
```

图7 mixed格式和now()

可以看到，MySQL用的居然是`statement`格式。你一定会奇怪，如果这个`binlog`过了1分钟才传给备库的话，那主备的数据不就不一致了吗？

接下来，我们再用`mysqlbinlog`工具来看看：

```
BEGIN  
/*!*/;  
# at 2825  
#181230 1:11:31 server id 1 end_log_pos 2942 CRC32 0x0ecd5082      Query   thread_id=4      exec_time=0      error_code=0  
SET TIMESTAMP=1546103491/*!*/;  
insert into t values(100, 1, now())  
/*!*/;  
# at 2942  
#181230 1:11:31 server id 1 end_log_pos 2973 CRC32 0x09877081      Xid = 41  
COMMIT/*!*/;
```

图8 TIMESTAMP 命令

从图中的结果可以看到，原来binlog在记录event的时候，多记了一条命令：SET TIMESTAMP=1546103491。它用 SET TIMESTAMP命令约定了接下来的now()函数的返回时间。

因此，不论这个binlog是1分钟之后被备库执行，还是3天后用来恢复这个库的备份，这个insert语句插入的行，值都是固定的。也就是说，通过这条SET TIMESTAMP命令，MySQL就确保了主备数据的一致性。

我之前看过有人在重放binlog数据的时候，是这么做的：用mysqlbinlog解析出日志，然后把里面的statement语句直接拷贝出来执行。

你现在知道了，这个方法是有风险的。因为有些语句的执行结果是依赖于上下文命令的，直接执行的结果很可能是错误的。

所以，用binlog来恢复数据的标准做法是，用 mysqlbinlog工具解析出来，然后把解析结果整个发给MySQL执行。类似下面的命令：

```
mysqlbinlog master.000001 --start-position=2738 --stop-position=2973 | mysql -h127.0.0.1 -P13000 -u$user -p$pwd
```

这个命令的意思是，将 master.000001 文件里面从第2738字节到第2973字节中间这段内容解析出来，放到MySQL去执行。

循环复制问题

通过上面对MySQL中binlog基本内容的理解，你现在可以知道，binlog的特性确保了在备库执行相同的binlog，可以得到与主库相同的状态。

因此，我们可以认为正常情况下主备的数据是一致的。也就是说，图1中A、B两个节点的内容是一致的。其实，图1中我画的是M-S结构，但实际生产上使用比较多的是双M结构，也就是图9所示的主备切换流程。

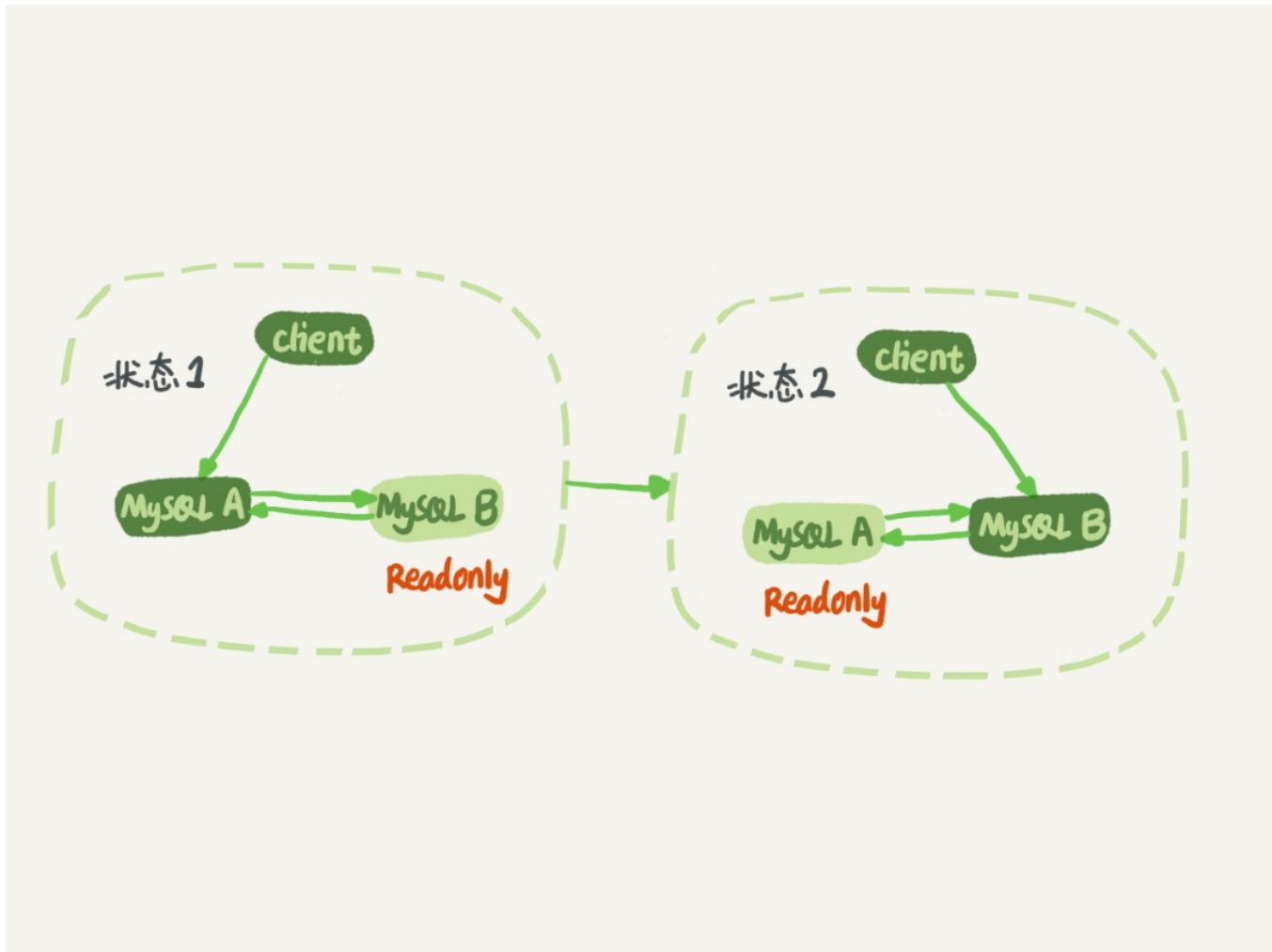


图 9 MySQL主备切换流程—双M结构

对比图9和图1，你可以发现，双M结构和M-S结构，其实区别只是多了一条线，即：节点A和B之间总是互为主备关系。这样在切换的时候就不用再修改主备关系。

但是，双M结构还有一个问题需要解决。

业务逻辑在节点A上更新了一条语句，然后再把生成的binlog发给节点B，节点B执行完这条更新语句后也会生成binlog。（我建议你把参数`log_slave_updates`设置为on，表示备库执行`relay log`后生成binlog）。

那么，如果节点A同时是节点B的备库，相当于又把节点B新生成的binlog拿过来执行了一次，然后节点A和B间，会不断地循环执行这个更新语句，也就是循环复制了。这个要怎么解决呢？

从上面的图6中可以看到，MySQL在binlog中记录了这个命令第一次执行时所在实例的server id。因此，我们可以用下面的逻辑，来解决两个节点间的循环复制的问题：

1. 规定两个库的server id必须不同，如果相同，则它们之间不能设定为主备关系；
2. 一个备库接到binlog并在重放的过程中，生成与原binlog的server id相同的新的binlog；
3. 每个库在收到从自己的主库发过来的日志后，先判断server id，如果跟自己的相同，表示这

个日志是自己生成的，就直接丢弃这个日志。

按照这个逻辑，如果我们设置了双M结构，日志的执行流就会变成这样：

1. 从节点A更新的事务，binlog里面记的都是A的server id；
2. 传到节点B执行一次以后，节点B生成的binlog 的server id也是A的server id；
3. 再传回给节点A，A判断到这个server id与自己的相同，就不会再处理这个日志。所以，死循环在这里就断掉了。

小结

今天这篇文章，我给你介绍了MySQL binlog的格式和一些基本机制，是后面我要介绍的读写分离等系列文章的背景知识，希望你可以认真消化理解。

binlog在MySQL的各种高可用方案上扮演了重要角色。今天介绍的可以说是所有MySQL高可用方案的基础。在这之上演化出了诸如多节点、半同步、MySQL group replication等相对复杂的方法。

我也跟你介绍了MySQL不同格式binlog的优缺点，和设计者的思考。希望你在做系统开发时候，也能借鉴这些设计思想。

最后，我给你留下一个思考题吧。

说到循环复制问题的时候，我们说MySQL通过判断server id的方式，断掉死循环。但是，这个机制其实并不完备，在某些场景下，还是有可能出现死循环。

你能构造出一个这样的场景吗？又应该怎么解决呢？

你可以把你的设计和分析写在评论区，我会在下一篇文章跟你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期我留给你的问题是，你在什么时候会把线上生产库设置成“非双1”。我目前知道的场景，有以下这些：

1. 业务高峰期。一般如果有预知的高峰期，DBA会有预案，把主库设置成“非双1”。
2. 备库延迟，为了让备库尽快赶上主库。[@永恒记忆](#)和[@Second Sight](#)提到了这个场景。
3. 用备份恢复主库的副本，应用binlog的过程，这个跟上一种场景类似。
4. 批量导入数据的时候。

一般情况下，把生产库改成“非双1”配置，是设置`innodb_flush_logs_at_trx_commit=2`、`sync_binlog=1000`。

评论区留言点赞板：

@way 同学提到了一个有趣的现象，由于从库设置了`binlog_group_commit_sync_delay`和`binlog_group_commit_sync_no_delay_count`导致一直延迟的情况。我们在主库设置这两个参数，是为了减少`binlog`的写盘压力。备库这么设置，尤其在“快要追上”的时候，就反而会受这两个参数的拖累。一般追主备就用“非双1”（追上记得改回来）。

@一大只 同学验证了在`sync_binlog=0`的情况下，设置`sync_delay`和`sync_no_delay_count`的现象，点赞这种发现边界的意识和手动验证的好习惯。是这样的：`sync_delay`和`sync_no_delay_count`的逻辑先走，因此该等还是会等。等到满足了这两个条件之一，就进入`sync_binlog`阶段。这时候如果判断`sync_binlog=0`，就直接跳过，还是不调`fsync`。

@锅子 同学提到，设置`sync_binlog=0`的时候，还是可以看到`binlog`文件马上做了修改。这个是对的，我们说“写到了`page cache`”，就是文件系统的`page cache`。而你用`ls`命令看到的就是文件系统返回的结果。

The image is a promotional graphic for a MySQL course. It features a portrait of Ding Qi, a man with short dark hair and glasses, wearing a black button-down shirt, standing with his arms crossed. To his left is the title "MySQL 实战 45 讲" in large, bold, dark font, with the subtitle "从原理到实战，丁奇带你搞懂 MySQL" below it. To the far left is the Geektime logo (a stylized orange 'G'). Below the title, the teacher's name "林晓斌" is listed, followed by "网名丁奇" and "前阿里资深技术专家". At the bottom, there is a call-to-action: "新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。"

精选留言



Sinyo

2

主库 A 从本地读取 binlog，发给从库 B；

老师，请问这里的本地是指文件系统的 page cache 还是 disk 呢？

2019-01-21

| 作者回复

好问题，

是这样的，对于 A 的线程来说，就是“读文件”，

1. 如果这个文件现在还在 page cache 中，那就最好了，直接读走；

2. 如果不在 page cache 里，就只好去磁盘读

这个行为是文件系统控制的，MySQL 只是执行“读文件”这个操作

2019-01-21



Leon

1

老师，我想问下双M架构下，主从复制，是不是一方判断自己的数据比对方少就从对方复制，判断依据是什么

2019-01-25

| 作者回复

好问题。

一开始创建主备关系的时候，是由备库指定的。

比如基于位点的主备关系，备库说“我要从 binlog 文件 A 的位置 P”开始同步，主库就从这个指定的位置开始往后发。

而主备复制关系搭建完成以后，是主库来决定“要发数据给备库”的。

所以主库有生成新的日志，就会发给备库。

2019-01-25



观弈道人

6

老师你好，问个备份问题，假如周日 23 点做了备份，周二 20 点需要恢复数据，那么在用 binlog 恢复时，如何恰好定位到周日 23 点的 binlog，谢谢。

2019-01-07

| 作者回复

Mysqlbinlog 有个参数—stop-datetime

2019-01-07



堕落天使

3

老师，您好，问一个关于 change buffer 的问题。

对于 insert 语句来说，change buffer 的优化主要在非唯一的二级索引上，因为主键是唯一索引，插入必须要判断是否存在。

那么对于 update 语句呢？如下（假设 c 有非唯一索引，id 是主键，d 没有索引）：

```
update t set d=2 where c=10;
```

原先以为：从索引c取出id之后，不会回表，也不会把修改行的数据读入内存，而是直接在change buffer中记录一下。但看了今天得内容之后又迷糊了，因为如果不把修改行的数据读入内存，它又怎么把旧数据写入binlog中呢？

所以我想问的就是，上面的sql语句会不会把修改行的内容也读进内存？如果读进内存，那读进内存的这一步难道就为了写binlog吗？如果不读进内存，那binlog中的旧数据又是怎么来的呢？还有delete语句也同理。

2019-01-07

作者回复

修改的行要读入内存呀

写binlog只需要主键索引上的值

你这个语句的话，如果字段c d上都有索引，那么c用不上chsnge buffer，

D可能可以同上

2019-01-07



hua168

2

大神，我前些天去面试，面试官问了一题：

mysql做主从，一段时间后发现从库在高峰期会发生一两条条数据丢失（不记得是查询行空白还是查询不到了），主从正常，怎么判断？

1.我问他是不是所以从库都是一样，他说不一样

2.我说低峰期重做新的从库观察，查看日志有没有报错？他好像不满意这个答案。

二、他还问主库挂了怎么办？

1. mysql主从+keepalived/heartbeat

有脑裂，还是有前面丢数据问题

2. 用MMM或HMA之类

3.用ZK之类

三、写的压力大怎么办？

我回答，分库，分表

感觉整天他都不怎么满意，果然没让我复试了，我郁闷呀，我就面试运维的，问数据这么详细。』

大神，能说下我哪里有问题吗？现在我都想不明白』

2019-01-08

作者回复

运维现在要求也挺高的

第一个问题其实我也没看懂，“高峰期丢数据”是指主备延迟查不到数据，还是真的丢了，得先问清楚下

不过你回答的第二点不太好，低峰期重做这个大家都知道要这么做，而且只是修复动作，没办法用来定位原因，面试官是要问你分析问题的方法（方向错误）
重搭从库错误日志里面什么都没有的（这个比较可惜，暴露了对字节不够了解，一般不了解的方法在面试的时候是不如不说的）

第二个问题三点都是你回答的吗？那还算回答得可以的，但是不能只讲名词，要找个你熟悉细节的方案展开一下

三方向也是对的

我估计就是第一个问题减分比较厉害

2019-01-08



HuaMax

2

课后题。如果在同步的过程中修改了server id，那用原server id 生成的log被两个M认为都不是自己的而被循环执行，不知这种情况会不会发生

2019-01-07

| 作者回复

是的，会

2019-01-07



风二中

1

在主库执行这条 SQL 语句的时候，用的是索引 a；而在备库执行这条 SQL 语句的时候，却使用了索引 t_modified

老师，您好，这里索引选择不一样，是因为前面提到的mysql 会选错索引吗？这种情况应该发生比较少吧，这里应该都会选择索引a吧，还是说这里只是一个事例，还有更复杂的情况

2019-01-12

| 作者回复

对，只是一个举例的

2019-01-12



夜空中最亮的星 (华仔)

1

级联复制，3个数据库，首尾相连，应会出现死循环

2019-01-08

| 作者回复

不会哦，1给2，2给3，3给1，1就放弃了

不过引入第三个节点的思路是对的哈

2019-01-08



changshan

1

老师好，**mixed**是**row**和**statement**的优点整合折中方案，这应该是好多系统设计理念吧？那么问题一：**mixed**既然能判断是什么时候使用**row**，什么时候使用**statement**，那么为什么好多推荐都是使用**row**而且不是使用**mixed**呢？是因为**mixed**这种模式下的自动选择转换不准确可能会出现主从问题吗？问题二：当使用**mixed**模式情况下，**mysql**内部是怎么判断的呢？比如有**limit**语句就会选择记录**row**格式，有**now()**函数还是会记录**statement**格式，**mysql**只是简单的某些特定场景下会使用记录**row**格式吗？谢谢。

2019-01-07

作者回复

1. 就是我们文中后面说的那些原因，要用这些**binlog**的内容去做别的事情
2. 对，固定模式下的。好问题，我去拉不下最新版本代码看下规则

2019-01-07



柚子

1

大佬您好，文中说现在越来越多的使用**row**方式的**binlog**，那么只能选择接受写入慢和占用空间大的弊端么？

2019-01-07

作者回复

是的，当然还有**minimal**可选，会好些

2019-01-07



汪炜

0

老师，问个问题，希望能被回答：

mmysql不是双一设置的时候，破坏了二阶段提交，事务已提交，**redo**没有及时刷盘，**binlog**刷盘了，这种情况，**mysql**是怎么恢复的，这个事务到底算不算提交？

2019-01-23

作者回复

如果“**redo**没有及时刷盘，**binlog**刷盘了”之后瞬间数据库所在主机掉电，

主机重启，**MySQL**重启以后，这个事务会丢失；这里确实会引起日志和数据不一致，这个就是我们说要默认设置为双1的原因之一哈

2019-01-23



Mackie .Weng

0

老师，你的课真好，你讲的都是生产实际用到的，点赞~

不过近期有点苦恼，要请教一下近期遇到的事

场景：

SSD硬盘，我们数据一天一备份，想通过昨天凌晨备份+**binlog**恢复到最新数据，导出的**binlog**为2G，然后发现导入**binlog**花费了4, 5小时，看了下**binlog**日志里面有很多这种信息

at 2492

```
#190108 17:08:38 server id 2 end_log_pos 2601 CRC32 0x8b0598ec Query thread_id=1227779
5 exec_time=0 error_code=0
SET TIMESTAMP=1546938518/*!*/;
BEGIN
```

```
/*!*/;  
# at 2601  
# at 2633  
# at 2919  
#190108 17:08:38 server id 2 end_log_pos 2950 CRC32 0x13806369 Xid = 1924155105  
COMMIT/*!*/;
```

问题：

- 1、在导出binlog为2G而且看了下里面很多这种事务，这是什么东西，有什么用吗
- 2、这种事务在导出binlog的时候可以不记录吗，然后来提高恢复数据的速度？
- 3、如果这是正常的情况，有无推荐更好的数据恢复方案或者工具

感谢老师

2019-01-14

作者回复

1. 有用，最好保留这些信息一起执行；
2. 提升不了多少速度的，花时间主要还是在更新数据的那些日志上，那些日志又不能去掉的：）
3. 这个方案是串行恢复。你可以把全量恢复出来的库，接成线上一个从库的备库，开并行复制，

2019-01-14



秋一匹

0

老师，您好。我这慢了一步。。。学习晚了点。我这之前碰到了个问题，有一段时间主从复制延迟比较厉害，达到5s左右吧，一般都是1~2秒吧。首先排除不是网络原因。想问下还有哪些因素会影响主从复制呢？

2019-01-10

作者回复

还是要给一下更具体的信息

比如主库的tps

备库的跟复制相关的配置等信息

2019-01-10



Mr.Strive.Z.H.L

0

老师你好：

有一个疑惑，多条语句同时在commit阶段过程中，如果发生写入binlog和写入redolog的顺序不一致的情况。主从备份的时候，从库是不是会导致数据不一致呀？

2019-01-10



未知

0

老师在讲row模式的数据恢复时，感觉insert, update, delete的数据格式和undo log的差不多。之前文章一直说redo和binlog，老师抽空也讲下undo和回滚段的知识。

2019-01-10

作者回复

Undo前面大致有说过了，你要了解undo的什么内容呢

2019-01-10



光

0

林老师今天遇到个问题就是主从同步延迟，查到主从状态中出现：`Slave_SQL_Running_State: Waiting for Slave Workers to free pending events`。不知道这个是否会引起延迟。查了些资料说得都不是很明白。老师是否可以简短解答下。以及这种延迟如何避免。

2019-01-09

作者回复

这个的意思是，现在工作线程里面等待的队列太多，都已经超过上限了，要等工作线程消化掉一些事务再分

简单说，就是备库的应用日志的队列太慢了。。

2019-01-10



梁中华

0

我有一个比较极端一点的HA问题，假设主库的binlog刚写成功还未来得及把binlog同步到从库，主库就掉电了，这时候从库的数据会不完整吗？

第二个问题，原主库重启加入集群后，那条没有传出去的binlog会如何处理？

2019-01-09



不迷失

0

请教一下，生产环境能不能使用正常使用表连接？要注意哪些地方？DBA总是说不建议用，还催促我将使用了表连接的地方改造，但也说不出个所以然。目前在两个百万级数据的表中有用到内连接，并没有觉得有什么问题

2019-01-08

作者回复

索引使用正确，不要出现全表扫描，其实OK的

2019-01-08



hua168

0

这样，我是想增加一些经验，怕后面面试又遇到，想问一下大神分析思路，这种问题没经验回答。

我就差点回答用阿里云的DRDS了】

现在开源的mysql中间件生存环境中用什么比较多呀？mycat还是网易的cetus？

海量存储阿里云有OSS，有没有对应的开源软件呀？用于生存环境的，没有接触过，想问下，搞下实验后再去找工作。】

2019-01-08

作者回复

中间件作为练手这些都可以的

你搜“开源分布式存储系统”

2019-01-08



React

0

老师好，文章前面说主从最好从机设置**readonly**.那么在双主的情况(互为主备)下，设置不同的自增值是否就可以不用设置只读了？且此时复制是否可以跳过主键冲突，因为自增值不同？

2019-01-08

| 作者回复

如果自增值严格控制了，也没必要设置跳过主键冲突了对吧（反正不冲突）

除非你的业务就是设计好支持多点写入，否则还是把不写入的都设置上**readonly**吧

2019-01-08

25 | MySQL是怎么保证高可用的？

2019-01-09 林晓斌



在上一篇文章中，我和你介绍了**binlog**的基本内容，在一个主备关系中，每个备库接收主库的**binlog**并执行。

正常情况下，只要主库执行更新生成的所有**binlog**，都可以传到备库并被正确地执行，备库就能达到跟主库一致的状态，这就是最终一致性。

但是，MySQL要提供高可用能力，只有最终一致性是不够的。为什么这么说呢？今天我就着重和你分析一下。

这里，我再放一次上一篇文章中讲到的双M结构的主备切换流程图。

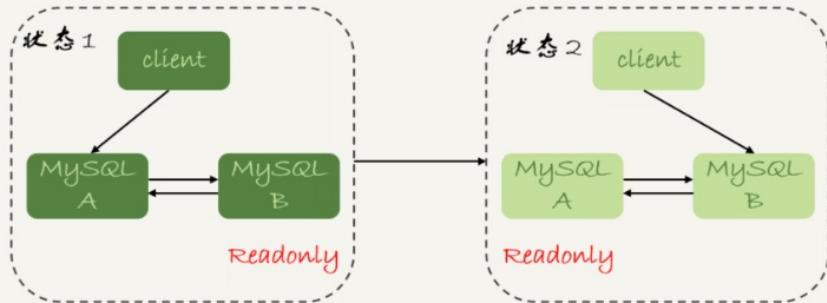


图 1 MySQL主备切换流程—双M结构

主备延迟

主备切换可能是一个主动运维动作，比如软件升级、主库所在机器按计划下线等，也可能是被动操作，比如主库所在机器掉电。

接下来，我们先一起看看主动切换的场景。

在介绍主动切换流程的详细步骤之前，我要先跟你说明一个概念，即“同步延迟”。与数据同步有关的时间点主要包括以下三个：

1. 主库A执行完成一个事务，写入binlog，我们把这个时刻记为T1；
2. 之后传给备库B，我们把备库B接收完这个binlog的时刻记为T2；
3. 备库B执行完成这个事务，我们把这个时刻记为T3。

所谓主备延迟，就是同一个事务，在备库执行完成的时间和主库执行完成的时间之间的差值，也就是T3-T1。

你可以在备库上执行`show slave status`命令，它的返回结果里面会显示

`seconds_behind_master`, 用于表示当前备库延迟了多少秒。

`seconds_behind_master`的计算方法是这样的：

1. 每个事务的`binlog`里面都有一个时间字段，用于记录主库上写入的时间；
2. 备库取出当前正在执行的事务的时间字段的值，计算它与当前系统时间的差值，得到`seconds_behind_master`。

可以看到，其实`seconds_behind_master`这个参数计算的就是 $T3-T1$ 。所以，我们可以用`seconds_behind_master`来作为主备延迟的值，这个值的时间精度是秒。

你可能会问，如果主备库机器的系统时间设置不一致，会不会导致主备延迟的值不准？

其实不会的。因为，备库连接到主库的时候，会通过执行`SELECT UNIX_TIMESTAMP()`函数来获得当前主库的系统时间。如果这时候发现主库的系统时间与自己不一致，备库在执行`seconds_behind_master`计算的时候会自动扣掉这个差值。

需要说明的是，在网络正常的时候，日志从主库传给备库所需的时间是很短的，即 $T2-T1$ 的值是非常小的。也就是说，网络正常情况下，主备延迟的主要来源是备库接收完`binlog`和执行完这个事务之间的时间差。

所以说，主备延迟最直接的表现是，备库消费中转日志（`relay log`）的速度，比主库生产`binlog`的速度要慢。接下来，我就和你一起分析下，这可能是由哪些原因导致的。

主备延迟的来源

首先，有些部署条件下，备库所在机器的性能要比主库所在的机器性能差。

一般情况下，有人这么部署时的想法是，反正备库没有请求，所以可以用差一点儿的机器。或者，他们会把20个主库放在4台机器上，而把备库集中在一台机器上。

其实我们都知道，更新请求对IOPS的压力，在主库和备库上是无差别的。所以，做这种部署时，一般都会将备库设置为“非双1”的模式。

但实际上，更新过程中也会触发大量的读操作。所以，当备库主机上的多个备库都在争抢资源的时候，就可能会导致主备延迟了。

当然，这种部署现在比较少了。因为主备可能发生切换，备库随时可能变成主库，所以主备库选用相同规格的机器，并且做对称部署，是现在比较常见的情况。

追问1：但是，做了对称部署以后，还可能会有延迟。这是为什么呢？

这就是第二种常见的可能了，即备库的压力大。一般的看法是，主库既然提供了写能力，那么

备库可以提供一些读能力。或者一些运营后台需要的分析语句，不能影响正常业务，所以只能在备库上跑。

我真就见过不少这样的情况。由于主库直接影响业务，大家使用起来会比较克制，反而忽视了备库的压力控制。结果就是，备库上的查询耗费了大量的CPU资源，影响了同步速度，造成主备延迟。

这种情况，我们一般可以这么处理：

1. 一主多从。除了备库外，可以多接几个从库，让这些从库来分担读的压力。
2. 通过binlog输出到外部系统，比如Hadoop这类系统，让外部系统提供统计类查询的能力。

其中，一主多从的方式大都会被采用。因为作为数据库系统，还必须保证有定期全量备份的能力。而从库，就很适合用来做备份。

备注：这里需要说明一下，从库和备库在概念上其实差不多。在我们这个专栏里，为了方便描述，我把会在HA过程中被选成新主库的，称为备库，其他的称为从库。

追问2：采用了一主多从，保证备库的压力不会超过主库，还有什么情况可能导致主备延迟吗？

这就是第三种可能了，即大事务。

大事务这种情况很好理解。因为主库上必须等事务执行完成才会写入binlog，再传给备库。所以，如果一个主库上的语句执行10分钟，那这个事务很可能就会导致从库延迟10分钟。

不知道你所在公司的DBA有没有跟你这么说过：不要一次性地用**delete**语句删除太多数据。其实，这就是一个典型的大事务场景。

比如，一些归档类的数据，平时没有注意删除历史数据，等到空间快满了，业务开发人员要一次性地删掉大量历史数据。同时，又因为要避免在高峰期操作会影响业务（至少有这个意识还是很不错的），所以会在晚上执行这些大量数据的删除操作。

结果，负责的DBA同学半夜就会收到延迟报警。然后，DBA团队就要求你后续再删除数据的时候，要控制每个事务删除的数据量，分成多次删除。

另一种典型的大事务场景，就是大表**DDL**。这个场景，我在前面的文章中介绍过。处理方案就是，计划内的**DDL**，建议使用**gh-ost**方案（这里，你可以再回顾下第13篇文章[《为什么表数据删掉一半，表文件大小不变？》](#)中的相关内容）。

追问3：如果主库上也不做大事务了，还有什么原因会导致主备延迟吗？

造成主备延迟还有一个大方向的原因，就是**备库的并行复制能力**。这个话题，我会留在下一篇
文章再和你详细介绍。

其实还是有不少其他情况会导致主备延迟，如果你还碰到过其他场景，欢迎你在评论区给我留言，我来和你一起分析、讨论。

由于主备延迟的存在，所以在主备切换的时候，就相应的有不同的策略。

可靠性优先策略

在图1的双M结构下，从状态1到状态2切换的详细过程是这样的：

1. 判断备库B现在的seconds_behind_master，如果小于某个值（比如5秒）继续下一步，否则持续重试这一步；
2. 把主库A改成只读状态，即把readonly设置为true；
3. 判断备库B的seconds_behind_master的值，直到这个值变成0为止；
4. 把备库B改成可读写状态，也就是把readonly设置为false；
5. 把业务请求切到备库B。

这个切换流程，一般是由专门的HA系统来完成的，我们暂时称之为可靠性优先流程。

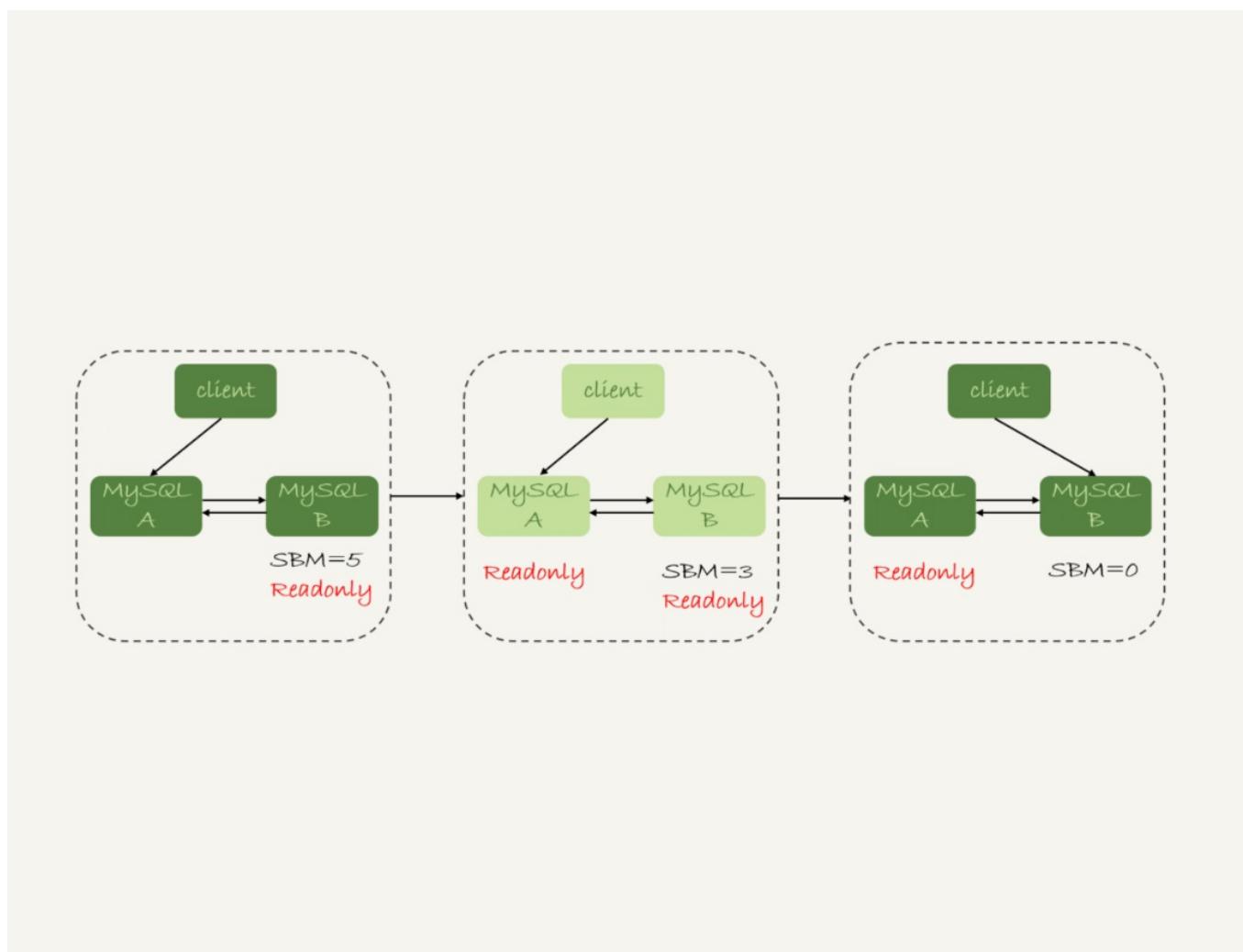


图2 MySQL可靠性优先主备切换流程

备注：图中的**SBM**，是seconds_behind_master参数的简写。

可以看到，这个切换流程中是有不可用时间的。因为在步骤2之后，主库A和备库B都处于**readonly**状态，也就是说这时系统处于不可写状态，直到步骤5完成后才能恢复。

在这个不可用状态中，比较耗费时间的是步骤3，可能需要耗费好几秒的时间。这也是为什么需要在步骤1先做判断，确保seconds_behind_master的值足够小。

试想如果一开始主备延迟就长达30分钟，而不先做判断直接切换的话，系统的不可用时间就会长达30分钟，这种情况一般业务都是不可接受的。

当然，系统的不可用时间，是由这个数据可靠性优先的策略决定的。你也可以选择可用性优先的策略，来把这个不可用时间几乎降为0。

可用性优先策略

如果我强行把步骤4、5调整到最开始执行，也就是说不等主备数据同步，直接把连接切到备库B，并且让备库B可以读写，那么系统几乎就没有不可用时间了。

我们把这个切换流程，暂时称作可用性优先流程。这个切换流程的代价，就是可能出现数据不一致的情况。

接下来，我就和你分享一个可用性优先流程产生数据不一致的例子。假设有一个表t：

```
mysql> CREATE TABLE `t` (
    `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
    `c` int(11) unsigned DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;

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

这个表定义了一个自增主键id，初始化数据后，主库和备库上都是3行数据。接下来，业务人员要继续在表t上执行两条插入语句的命令，依次是：

```
insert into t(c) values(4);
insert into t(c) values(5);
```

假设，现在主库上其他的数据表有大量的更新，导致主备延迟达到5秒。在插入一条c=4的语句

后，发起了主备切换。

图3是可用性优先策略，且`binlog_format=mixed`时的切换流程和数据结果。

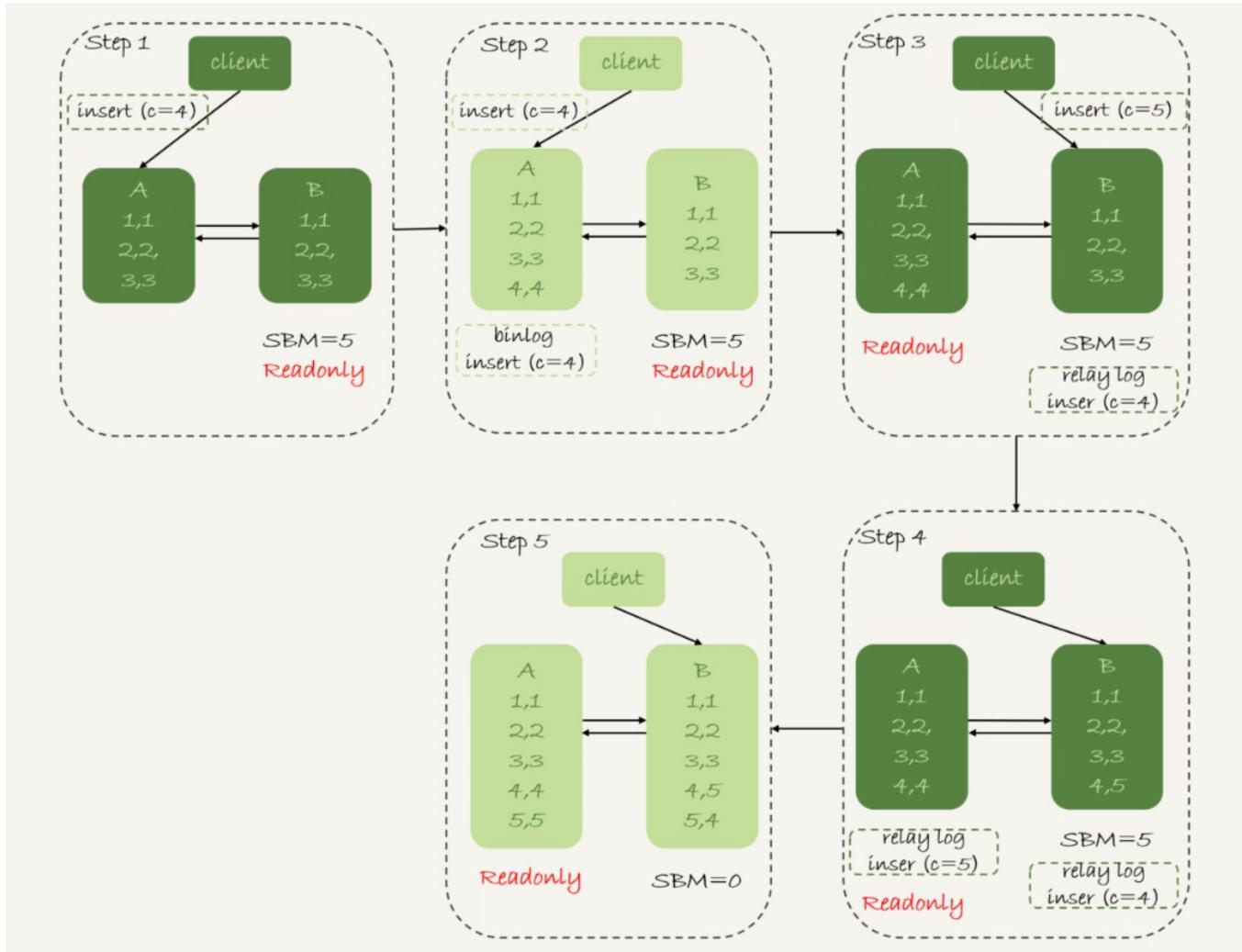


图3 可用性优先策略，且`binlog_format=mixed`

现在，我们一起分析下这个切换流程：

- 步骤2中，主库A执行完`insert`语句，插入了一行数据（4,4），之后开始进行主备切换。
- 步骤3中，由于主备之间有5秒的延迟，所以备库B还没来得及应用“插入c=4”这个中转日志，就开始接收客户端“插入 c=5”的命令。
- 步骤4中，备库B插入了一行数据（4,5），并且把这个`binlog`发给主库A。
- 步骤5中，备库B执行“插入c=4”这个中转日志，插入了一行数据（5,4）。而直接在备库B执行的“插入c=5”这个语句，传到主库A，就插入了一行新数据（5,5）。

最后的结果就是，主库A和备库B上出现了两行不一致的数据。可以看到，这个数据不一致，是由可用性优先流程导致的。

那么，如果我还是用可用性优先策略，但设置`binlog_format=row`，情况又会怎样呢？

因为**row**格式在记录**binlog**的时候，会记录新插入的行的所有字段值，所以最后只会有一行不一致。而且，两边的主备同步的应用线程会报错**duplicate key error**并停止。也就是说，这种情况下，备库B的(5,4)和主库A的(5,5)这两行数据，都不会被对方执行。

图4中我画出了详细过程，你可以自己再分析一下。

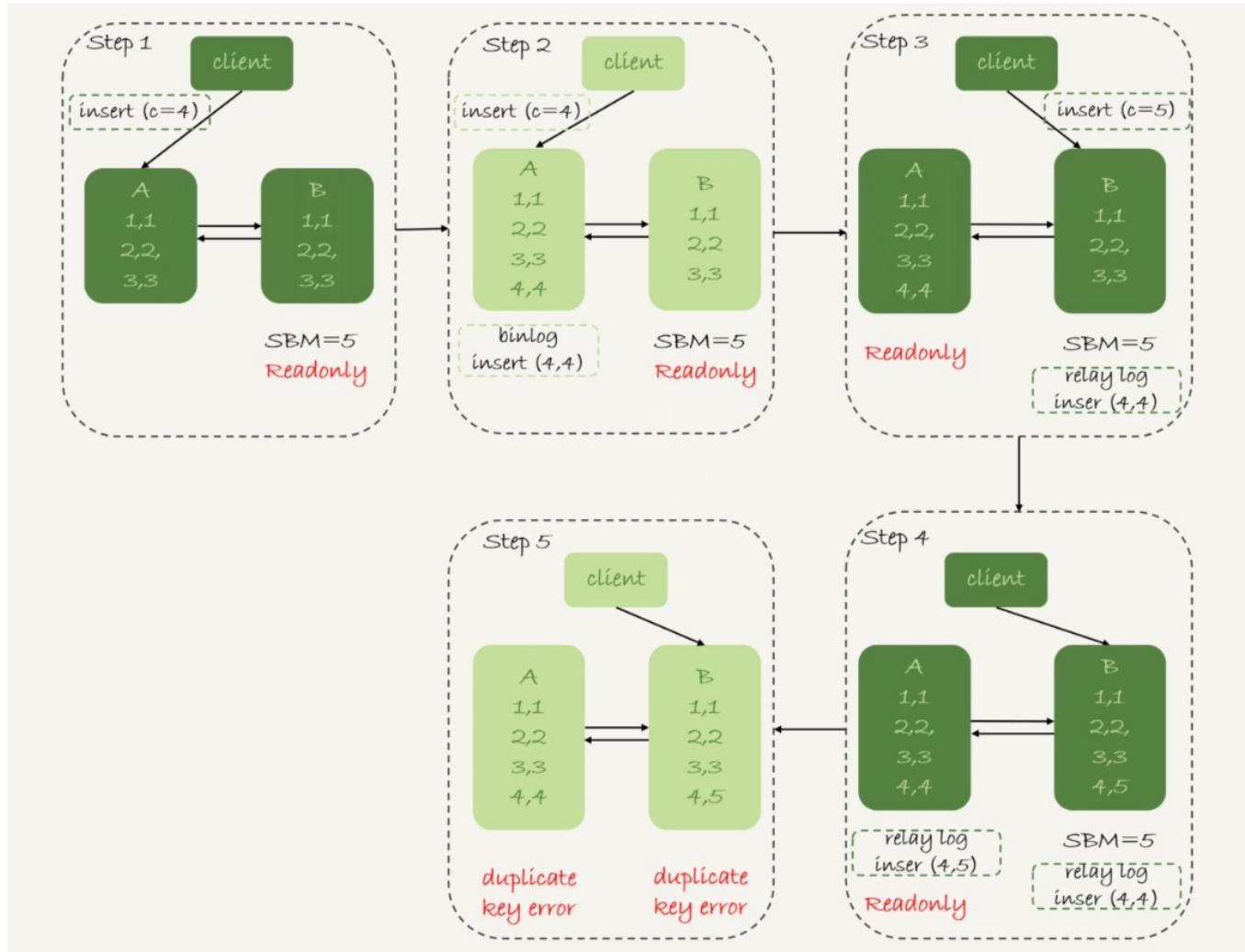


图4 可用性优先策略，且`binlog_format=row`

从上面的分析中，你可以看到一些结论：

1. 使用**row**格式的**binlog**时，数据不一致的问题更容易被发现。而使用**mixed**或者**statement**格式的**binlog**时，数据很可能悄悄地就不一致了。如果你过了很久才发现数据不一致的问题，很可能这时的数据不一致已经不可查，或者连带造成了更多的数据逻辑不一致。
2. 主备切换的可用性优先策略会导致数据不一致。因此，大多数情况下，我都建议你使用可靠性优先策略。毕竟对数据服务来说的话，数据的可靠性一般还是要优于可用性的。

但事无绝对，有没有哪种情况数据的可用性优先级更高呢？

答案是，有的。

我曾经碰到过这样的一个场景：

- 有一个库的作用是记录操作日志。这时候，如果数据不一致可以通过binlog来修补，而这个短暂的不一致也不会引发业务问题。
- 同时，业务系统依赖于这个日志写入逻辑，如果这个库不可写，会导致线上的业务操作无法执行。

这时候，你可能就需要选择先强行切换，事后再补数据的策略。

当然，事后复盘的时候，我们想到了一个改进措施就是，让业务逻辑不要依赖于这类日志的写入。也就是说，日志写入这个逻辑模块应该可以降级，比如写到本地文件，或者写到另外一个临时库里面。

这样的话，这种场景就又可以使用可靠性优先策略了。

接下来我们再看看，按照可靠性优先的思路，异常切换会是什么效果？

假设，主库A和备库B间的主备延迟是30分钟，这时候主库A掉电了，HA系统要切换B作为主库。我们在主动切换的时候，可以等到主备延迟小于5秒的时候再启动切换，但这时候已经别无选择了。

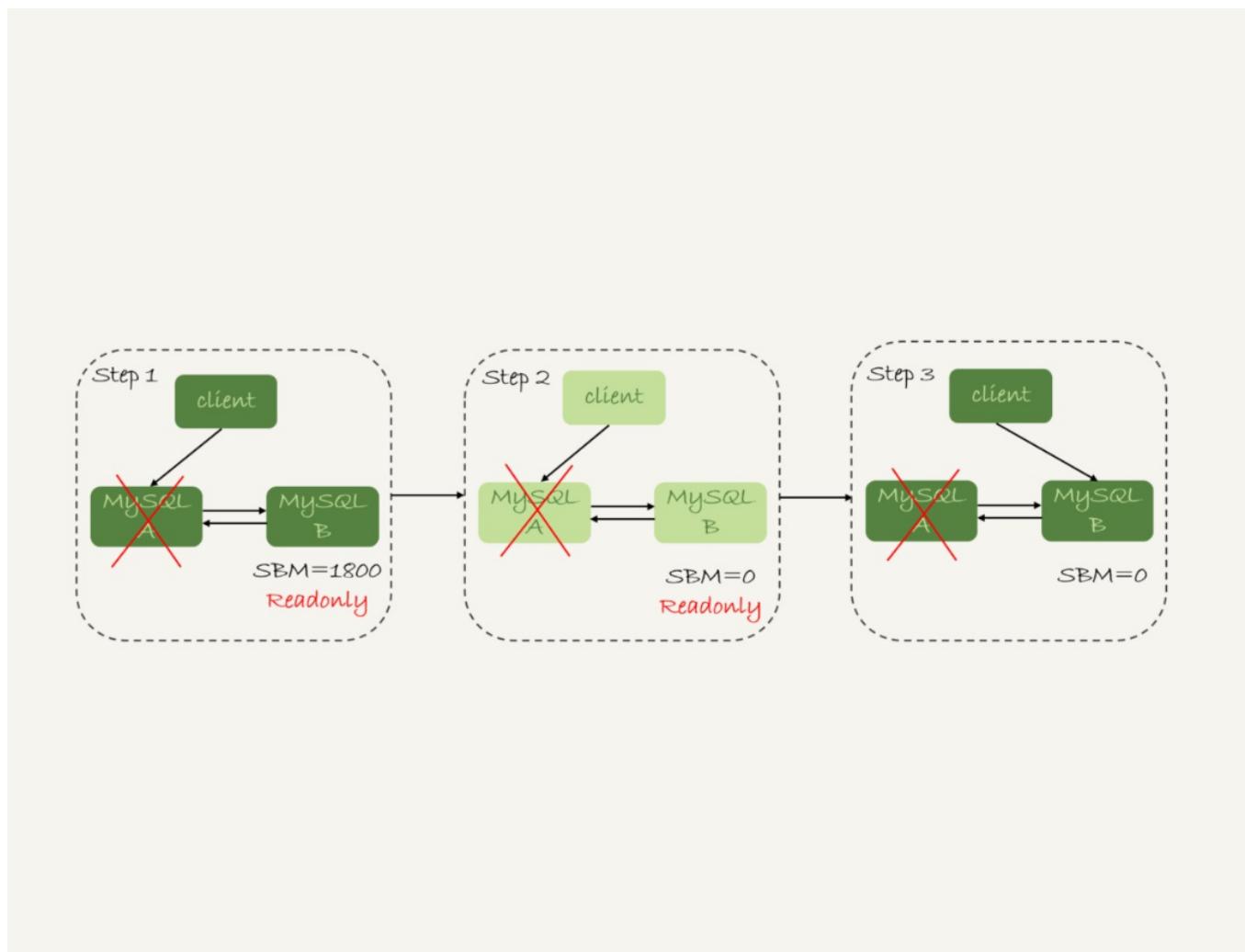


图5 可靠性优先策略，主库不可用

采用可靠性优先策略的话，你就必须得等到备库B的`seconds_behind_master=0`之后，才能切换。但现在的情况比刚刚更严重，并不是系统只读、不可写的问题了，而是系统处于完全不可用的状态。因为，主库A掉电后，我们的连接还没有切到备库B。

你可能会问，那能不能直接切换到备库B，但是保持B只读呢？

这样也不行。

因为，这段时间内，中转日志还没有应用完成，如果直接发起主备切换，客户端查询看不到之前执行完成的事务，会认为有“数据丢失”。

虽然随着中转日志的继续应用，这些数据会恢复回来，但是对于一些业务来说，查询到“暂时丢失数据的状态”也是不能被接受的。

聊到这里你就知道了，在满足数据可靠性的前提下，MySQL高可用系统的可用性，是依赖于主备延迟的。延迟的时间越小，在主库故障的时候，服务恢复需要的时间就越短，可用性就越高。

小结

今天这篇文章，我先和你介绍了MySQL高可用系统的基础，就是主备切换逻辑。紧接着，我又和你讨论了几种会导致主备延迟的情况，以及相应的改进方向。

然后，由于主备延迟的存在，切换策略就有不同的选择。所以，我又和你一起分析了可靠性优先和可用性优先策略的区别。

在实际的应用中，我更建议使用可靠性优先的策略。毕竟保证数据准确，应该是数据库服务的底线。在这个基础上，通过减少主备延迟，提升系统的可用性。

最后，我给你留下一个思考题吧。

一般现在的数据库运维系统都有备库延迟监控，其实就是在备库上执行`show slave status`，采集`seconds_behind_master`的值。

假设，现在你看到你维护的一个备库，它的延迟监控的图像类似图6，是一个45°斜向上的线段，你觉得可能是什么原因导致呢？你又会怎么去确认这个原因呢？

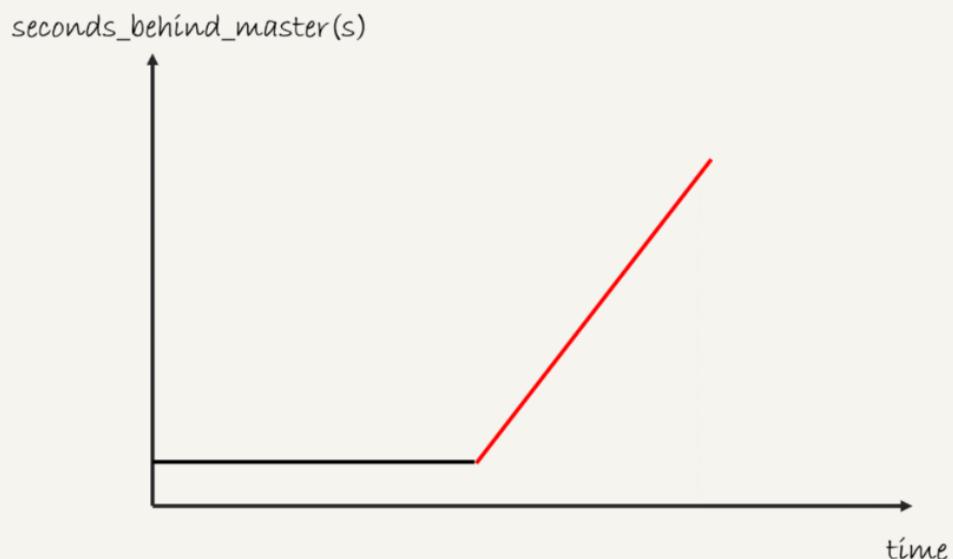


图6 备库延迟

你可以把你的分析写在评论区，我会在下一篇文章的末尾跟你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期我留给你的问题是，什么情况下双M结构会出现循环复制。

一种场景是，在一个主库更新事务后，用命令`set global server_id=x`修改了`server_id`。等日志再传回来的时候，发现`server_id`跟自己的`server_id`不同，就只能执行了。

另一种场景是，有三个节点的时候，如图7所示，`trx1`是在节点B执行的，因此binlog上的`server_id`就是B，binlog传给节点A，然后A和A'搭建了双M结构，就会出现循环复制。

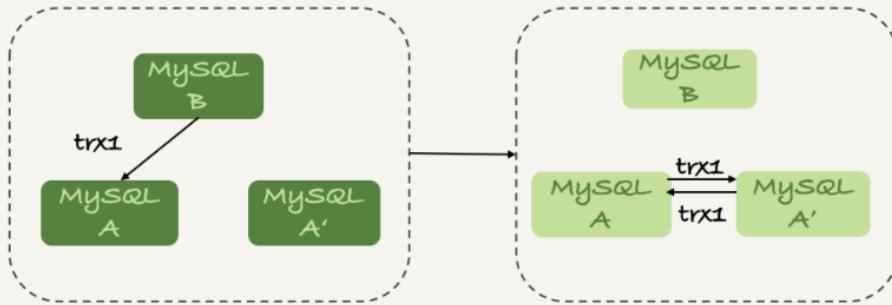


图7 三节点循环复制

这种三节点复制的场景，做数据库迁移的时候会出现。

如果出现了循环复制，可以在A或者A'上，执行如下命令：

```
stop slave;
CHANGE MASTER TO IGNORE_SERVER_IDS=(server_id_of_B);
start slave;
```

这样这个节点收到日志后就不会再执行。过一段时间后，再执行下面的命令把这个值改回来。

```
stop slave;
CHANGE MASTER TO IGNORE_SERVER_IDS=();
start slave;
```

评论区留言点赞板：

@一大只、@HuaMax 同学提到了第一个复现方法；

@Jonh同学提到了**IGNORE_SERVER_IDS**这个解决方法；

@React 提到，如果主备设置不同的步长，备库是不是可以设置为可读写。我的建议是，只要这个节点设计内就不会有业务直接在上面执行更新，就建议设置为**readonly**。

极客时间

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌 网名丁奇
前阿里资深技术专家

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

精选留言

-  某、人 12
- 遇到过下面几种造成主从延迟的情况：
1. 主库DML语句并发大,从库qps高
 2. 从库服务器配置差或者一台服务器上几台从库(资源竞争激烈,特别是io)
 3. 主库和从库的参数配置不一样
 4. 大事务(DDL,我觉得DDL也相当于一个大事务)
 5. 从库上在进行备份操作
 6. 表上无主键的情况(主库利用索引更改数据,备库回放只能用全表扫描,这种情况可以调整slave_rows_search_algorithms参数适当优化下)
 7. 设置的是延迟备库
 8. 备库空间不足的情况下

这期的问题：

看这曲线,应该是从库正在应用一个大事务,或者一个大表上无主键的情况(有该表的更新)

应该是T3随着时间的增长在增长,而T1这个时间点是没变的,造成的现象就是随着时间的增长,second_behind_master也是有规律的增长

2019-01-10

| 作者回复

分析的点很准确!

2019-01-11



undefined

2

问题答案:

1. 备库在执行复杂查询, 导致资源被占用
2. 备库正在执行一个大事务
3. DML 语句执行

老师我的理解对吗

2019-01-09

| 作者回复

1不太准确, 明天我会提到哈

23对的

2019-01-09



7号

1

老师, 生产环境有一张表需要清理, 该表大小140G。要保留最近一个月的数据, 又不能按时间直接用delete删(全表扫描), 本来想通过清空分区表删, 但是分区表又是哈希的。。有没好的办法呢?

2019-01-09

| 作者回复

估计下一个月占多少比例, 如果比较小就建新表, 把数据导过去吧

如果一个月占比高的话, 只能一点点删了。

时间字段有索引的话, 每个分区按时间过滤出来删除

2019-01-09



Sr7vy

1

问题1: T3的解释是: 备库执行完这个事物。则: Seconds_Behind_Master=T3-T1。如T1=30min, 主执行完成, 备没有执行。猜测1: 那么Seconds_Behind_Master=30min吗? 猜测2: 备执需要先把这个30min的事物执行完后, Seconds_Behind_Master=30min?

问题2: 很多时候是否能把Seconds_Behind_Master当作真正的延迟时间(面试常被问)? 如果能, pt-heartbeat存在还有啥意义啊?

2019-01-09

| 作者回复

问题1:

1. 备库没收到, 还是收到没执行, 前者0, 后者30

2. 第二问没看懂

问题2:

类似的，主库把日志都发给备库了吗

2019-01-09



万勇

1

主备同步延迟，工作中常遇到几种情况：

1. 主库做大量的dml操作，引起延迟
2. 主库有个大事务在处理，引起延迟
3. 对myisam存储引擎的表做dml操作，从库会有延迟。
4. 利用pt工具对主库的大表做字段新增、修改和添加索引等操作，从库会有延迟。

2019-01-09

作者回复



你是有故事的

2019-01-09



梁中华

1

我有一个比较极端一点的HA问题，假设主库的binlog刚写成功还未来得及把binlog同步到从库，主库就掉电了，这时候从库的数据会不完整吗？

第二个问题，原主库重启加入集群后，那条没有传出去的binlog会如何处理？

2019-01-09

作者回复

1. 可能会丢

2. 要看重启之后的拓扑结构了，如果还有节点是这个库的从库，还是会拿走的

2019-01-09



JJ

1

请问老师，主库断电了，怎么把binlog传给从库同步数据，怎么使的SBM为0主从切换呢？

2019-01-09

作者回复

等应用完就认为是SBM=0

如果不能接受主库有来不及传的，就使用semi-sync

2019-01-09



via

1

通过 binlog 输出到外部系统，比如 Hadoop 这类...

文中这个具体是可采用什么工具呢？

2019-01-09

作者回复

canal 可以了解下

2019-01-10



Sinyo

1

老师，在 binlog row 模式下，insert 到表中一条记录，这条记录中某个字段不填，该字段在表中有设置默认值，利用 canal 解析 binlog 出来，这个不填的字段会不存在；难道 binlog 只记录有插入的字段数据，表字段的默认数据就不会记录么？mysql 版本 5.7.22 canal 版本 1.0.3

2019-01-09

| 作者回复

不会啊

insert 记录的时候肯定都记录的

你的默认值是什么？

2019-01-10



700

0

老师请教下，MySQL 主从跨 IDC 的痛点是什么？同城 IDC 和异地 IDC 的痛点一样吗？怎么来解决这些痛点？

2019-01-22

| 作者回复

跨 IDC 还好吧，跨城市或者跨洲才比较麻烦

其实主要还是延迟的问题，这个确实不好解决。

业务开发的时候尽量是本城市访问，否则容易出现抖动

2019-01-23



强哥

0

今天跟公司的 dba 咨询了下，目前公司用的主备切换策略都是可用性优先，说是可靠性优先的话，可能会引起雪崩，主要还是业务的并发高，这种场景您是怎么看呢？麻烦给下思路谢谢。

2019-01-21

| 作者回复

额 那这个是要跟业务好好讨论一下架构设计的，可以这么跟业务说，如果是由于问题导致整个连不通，会不会雪崩？

也就是说，可用性不可能 100%，如果不可用就雪崩，表示架构需要优化。

之后才谈策略选择（否则这样根本没得谈哈）

2019-01-21



cheriston

0

老师 seconds_behind_master=0 也不能 100% 代表主库与从库之间没有延迟 吧？

2019-01-18

| 作者回复

嗯嗯，看下 28 篇

2019-01-18



悟空

0

老师文章末尾思考题部分，有点困惑求解答。

循环复制我之前理解是B->A->A'->B这样的拓扑结构。

而双M结构理解是A->A'->A，此时A是A'的主库和从库，B只能是A的从库。那么trx1在从库B上的更新就不会传给A。

文中是一种假设吗？还是我理解偏差了~

-----正文-----

trx1是在节点B执行的，因此binlog上的server_id就是B，binlog传给节点A，然后A和A'搭建了双M结构，就会出现循环复制。

2019-01-12

| 作者回复

这个说的是迁移过程，

也就是说，一开始A是B的从库，后来迁移过程中，停止了A和B的主备关系，让A和A'互为主备

2019-01-12



崔伟协

0

发生主从切换的时候，主有的最新数据没同步到从，会出现这种情况吗，出现了会怎么样

2019-01-11

| 作者回复

异常切换有可能的

要根据你的处理策略了，如果不能丢，有几个可选的

1. 不切换（等这个库自己恢复起来）

2. 使用semi-sync策略

3. 启动后业务做数据对账（这个一般用得少，成本高）

2019-01-11



任鹏斌

0

老师好发现我们系统中一条sql写法比较独特

SELECT

```
IF(ha.curricula_type = '02'  
AND ((cla.model_code IN ('CM05004', 'CM05001')  
AND cla.is_vip_video = 1)  
OR cla.model_code = 'CM05008'),  
'1',  
'0') AS 'isUK'  
FROM  
h_curricula ha,  
h_class cla  
WHERE  
ha.`code` = '2'
```

AND cla.`code` ='2'

使用查询分析器后所有列都无信息显示，只在Extra列显示，

Impossible WHERE noticed after reading const tables，不知道如何分析其执行计划。

如果code列均为两个表的主键类型是varchar，想知道这种情况下是否会产生笛卡尔积？

2019-01-11

| 作者回复

Impossible WHERE noticed after reading const tables

这个语句是不是执行结果是空？

2019-01-11



康磊

0

老师你好，现在一般采用读写分离，读的是从库，那么主从如果出现延迟的话，读库就读的不是最新数据，对这种问题有什么好建议吗？

2019-01-11

| 作者回复

第28篇专门讲这个问题，敬请期待！

2019-01-11



cyberty

0

请问老师，如果备库连接主库之后，主库的系统时间修改了，备库同步的时候是否会自动修正？

2019-01-10

| 作者回复

好问题，不会

2019-01-10



风萧雨瑟

0

老师问一下集群在开启并行复制的情况下：

主库参数：binlog_group_commit_sync_delay=1000; binlog_group_commit_sync_no_delay_count=10

从库：slave_parallel_type=LOGICAL_CLOCK; slave_parallel_workers=8

MySQL：社区版5.7.20

在从库上查看slave status的时Seconds_Behind_Master总是显示落后10-15，在有大量更新的情况下数据会一直增大，通过binlog来看的话Read_Master_Log_Pos 和Exec_Master_Log_Pos相差总是在1000+，甚至变大的更大。但将slave_parallel_type更改回默认值DATABASE时，Read_Master_Log_Pos 和Exec_Master_Log_Pos相差很小，甚至可以相同。

在不同的集群上开启并行复制都会出现相同的情况，但将slave_parallel_type更改回默认值DATABASE时都要比LOGICAL_CLOCK延迟情况要好。

更换5.7.24版本的情况下有同样的问题。

如果有两台从库，机器配置相同其它参数一样。一台设置成slave_parallel_type=DATABASE。而另一台设置成LOGICAL_CLOCK，不管是线上的表现还是通过sysbench压测来看，设置成LOGICAL_CLOCK的从库延迟确实要比DATABASE大一些。

这个情况从哪里排查一下？谢谢。

2019-01-10

作者回复

先看看**26**篇，然后再下面留下你的理解和新的疑问哈

2019-01-10



xm

0

一般主从延时多少算是合理的？是秒级别吗？

2019-01-10

作者回复

一般大于1就不好 ^_^

2019-01-10



Chris

0

老师，咨询个问题，现在遇到一个问题，**mysql**数据库总是**crash**，重新启动服务又正常，然后运行一段时间又会**crash**，报错如下**InnoDB: Assertion failure in thread 6792 in file fil0fil.cc line 5805**

InnoDB: Failing assertion: err == DB_SUCCESS

InnoDB: We intentionally generate a memory trap.

InnoDB: Submit a detailed bug report to <http://bugs.mysql.com>.

InnoDB: If you get repeated assertion failures or crashes, even

InnoDB: immediately after the mysqld startup, there may be

InnoDB: corruption in the InnoDB tablespace. Please refer to

<http://dev.mysql.com/doc/refman/5.7/en/forcing-innodb-recovery.html>

InnoDB: about forcing recovery.

08:52:33 UTC - mysqld got exception 0x80000003 ;

This could be because you hit a bug. It is also possible that this binary or one of the libraries it was linked against is corrupt, improperly built, or misconfigured. This error can also be caused by malfunctioning hardware.

Attempting to collect some information that could help diagnose the problem.

As this is a crash and something is definitely wrong, the information collection process might fail.

key_buffer_size=8388608

read_buffer_size=131072

max_used_connections=60

max_threads=151

thread_count=45

connection_count=45

It is possible that mysqld could use up to

key_buffer_size + (read_buffer_size + sort_buffer_size)*max_threads = 68010 K bytes of memory

Hope that's ok; if not, decrease some variables in the equation.

Thread pointer: 0x0

Attempting backtrace. You can use the following information to find out where mysqld died. If you see no messages after this, something went terribly wrong...

13f9b9812 mysqld.exe!my_sigabrt_handler()[my_thr_init.c:449]

13fd5e349 mysqld.exe!raise()[winsig.c:587]

13fd5d240 mysqld.exe!abort()[abort.c:82]

13fab9b08 mysqld.exe!ut_dbg_assertion_failed()[ut0dbg.cc:67]

13fae06da mysqld.exe!fil_aio_wait()[fil0fil.cc:5807]

13fa7eb84 mysqld.exe!io_handler_thread()

The manual page at <http://dev.mysql.com/doc/mysql/en/crashing>.

2019-01-10

| 作者回复

看着好像是磁盘问题了，你这个是5.7的那个小版本？

还有，尽量不要用windows系统哦！

2019-01-11

26 | 备库为什么会延迟好几个小时？

2019-01-11 林晓斌



在上一篇文章中，我和你介绍了几种可能导致备库延迟的原因。你会发现，这些场景里，不论是偶发性的查询压力，还是备份，对备库延迟的影响一般是分钟级的，而且在备库恢复正常以后都能够追上来。

但是，如果备库执行日志的速度持续低于主库生成日志的速度，那这个延迟就有可能成了小时级别。而且对于一个压力持续比较高的主库来说，备库很可能永远都追不上主库的节奏。

这就涉及到今天我要给你介绍的话题：备库并行复制能力。

为了便于你理解，我们再一起看一下第24篇文章 [《MySQL是怎么保证主备一致的？》](#) 的主备流程图。

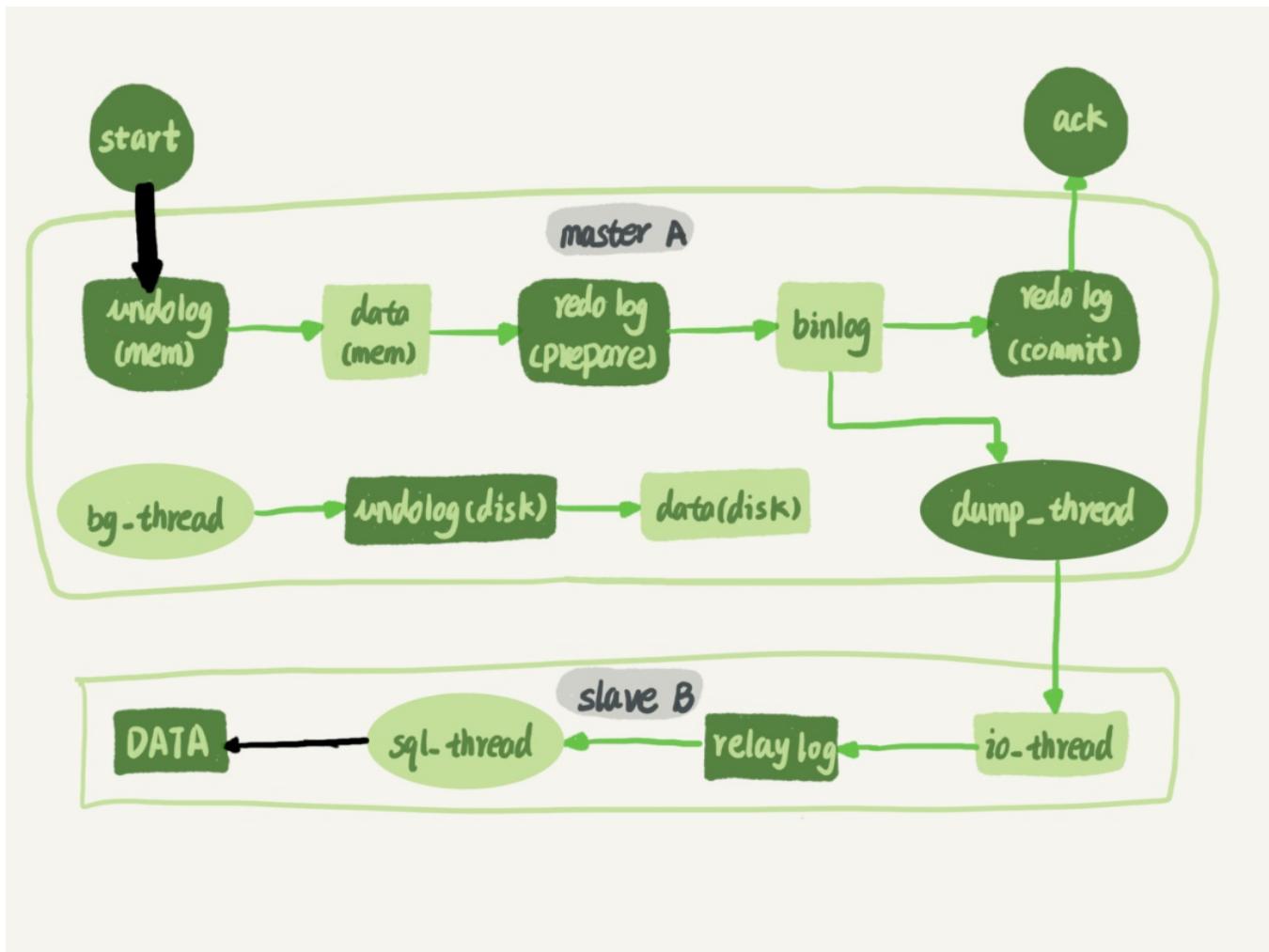


图1 主备流程图

谈到主备的并行复制能力，我们要关注的是图中黑色的两个箭头。一个箭头代表了客户端写入主库，另一箭头代表的是备库上`sql_thread`执行中转日志（`relay log`）。如果用箭头的粗细来代表并行度的话，那么真实情况就如图1所示，第一个箭头要明显粗于第二个箭头。

在主库上，影响并发度的原因就是各种锁了。由于InnoDB引擎支持行锁，除了所有并发事务都在更新同一行（热点行）这种极端场景外，它对业务并发度的支持还是很友好的。所以，你在性能测试的时候会发现，并发压测线程32就比单线程时，总体吞吐量高。

而日志在备库上的执行，就是图中备库上`sql_thread`更新数据(`DATA`)的逻辑。如果是用单线程的话，就会导致备库应用日志不够快，造成主备延迟。

在官方的5.6版本之前，MySQL只支持单线程复制，由此在主库并发高、TPS高时就会出现严重的主备延迟问题。

从单线程复制到最新版本的多线程复制，中间的演化经历了好几个版本。接下来，我就跟你说说MySQL多线程复制的演进过程。

其实说到底，所有的多线程复制机制，都是要把图1中只有一个线程的`sql_thread`，拆成多个线程，也就是都符合下面的这个模型：

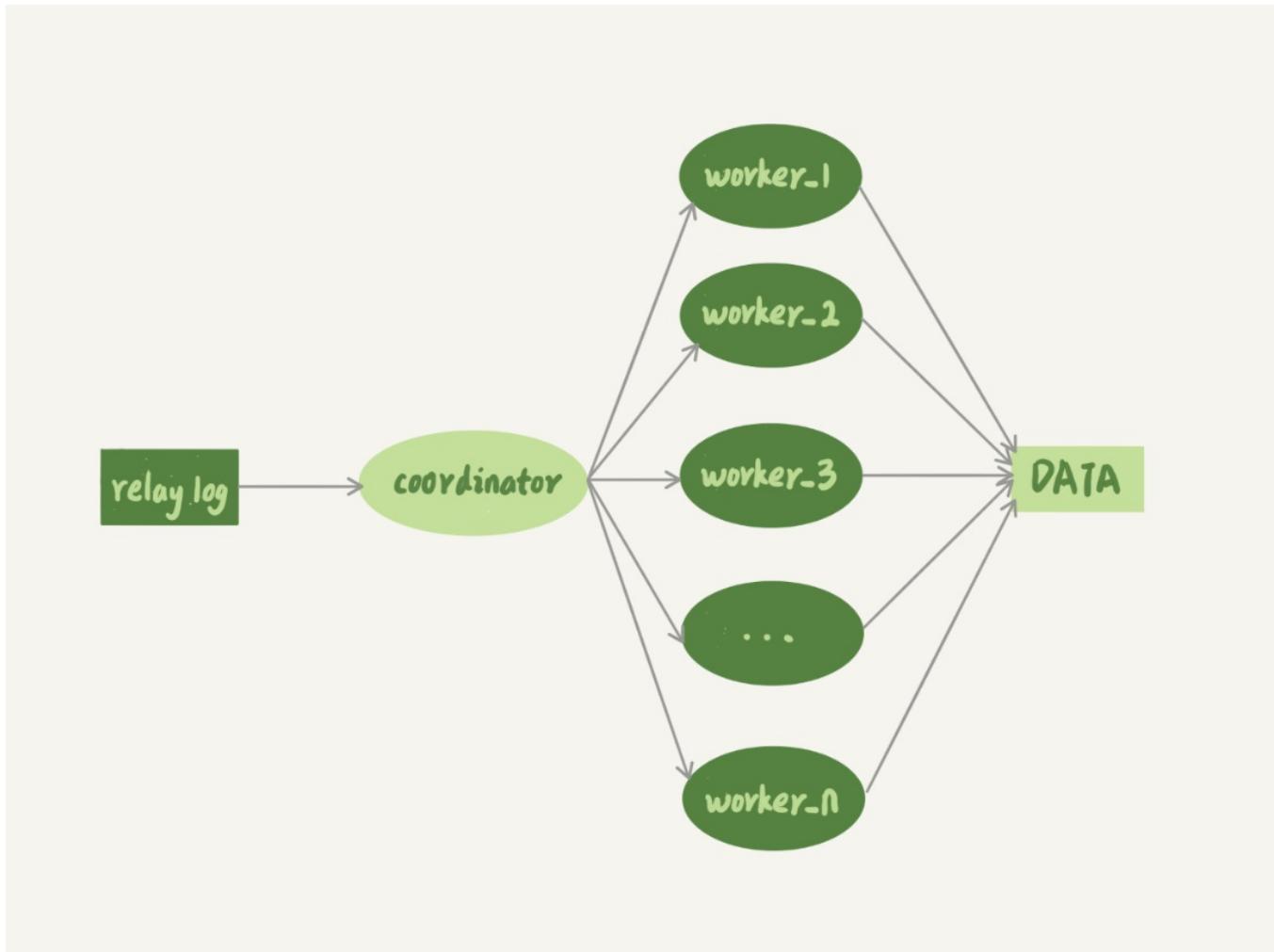


图2 多线程模型

图2中，**coordinator**就是原来的**sql_thread**，不过现在它不再直接更新数据了，只负责读取中转日志和分发事务。真正更新日志的，变成了**worker**线程。而**work**线程的个数，就是由参数 **slave_parallel_workers**决定的。根据我的经验，把这个值设置为8~16之间最好（32核物理机的情况），毕竟备库还有可能要提供读查询，不能把**CPU**都吃光了。

接下来，你需要先思考一个问题：事务能不能按照轮询的方式分发给各个**worker**，也就是第一个事务分给**worker_1**，第二个事务发给**worker_2**呢？

其实是不行的。因为，事务被分发给**worker**以后，不同的**worker**就独立执行了。但是，由于**CPU**的调度策略，很可能第二个事务最终比第一个事务先执行。而如果这时候刚好这两个事务更新的是同一行，也就意味着，同一行上的两个事务，在主库和备库上的执行顺序相反，会导致主备不一致的问题。

接下来，请你再设想一下另外一个问题：同一个事务的多个更新语句，能不能分给不同的**worker**来执行呢？

答案是，也不行。举个例子，一个事务更新了表**t1**和表**t2**中的各一行，如果这两条更新语句被分到不同**worker**的话，虽然最终的结果是主备一致的，但如果表**t1**执行完成的瞬间，备库上有一个查询，就会看到这个事务“更新了一半的结果”，破坏了事务逻辑的隔离性。

所以，**coordinator**在分发的时候，需要满足以下这两个基本要求：

1. 不能造成更新覆盖。这就要求更新同一行的两个事务，必须被分发到同一个**worker**中。
2. 同一个事务不能被拆开，必须放到同一个**worker**中。

各个版本的多线程复制，都遵循了这两条基本原则。接下来，我们就看看各个版本的并行复制策略。

MySQL 5.5版本的并行复制策略

官方MySQL 5.5版本是不支持并行复制的。但是，在2012年的时候，我自己服务的业务出现了严重的主备延迟，原因就是备库只有单线程复制。然后，我就先后写了两个版本的并行策略。

这里，我给你介绍一下这两个版本的并行策略，即按表分发策略和按行分发策略，以帮助你理解MySQL官方版本并行复制策略的迭代。

按表分发策略

按表分发事务的基本思路是，如果两个事务更新不同的表，它们就可以并行。因为数据是存储在表里的，所以按表分发，可以保证两个**worker**不会更新同一行。

当然，如果有跨表的事务，还是要把两张表放在一起考虑的。如图3所示，就是按表分发的规则。

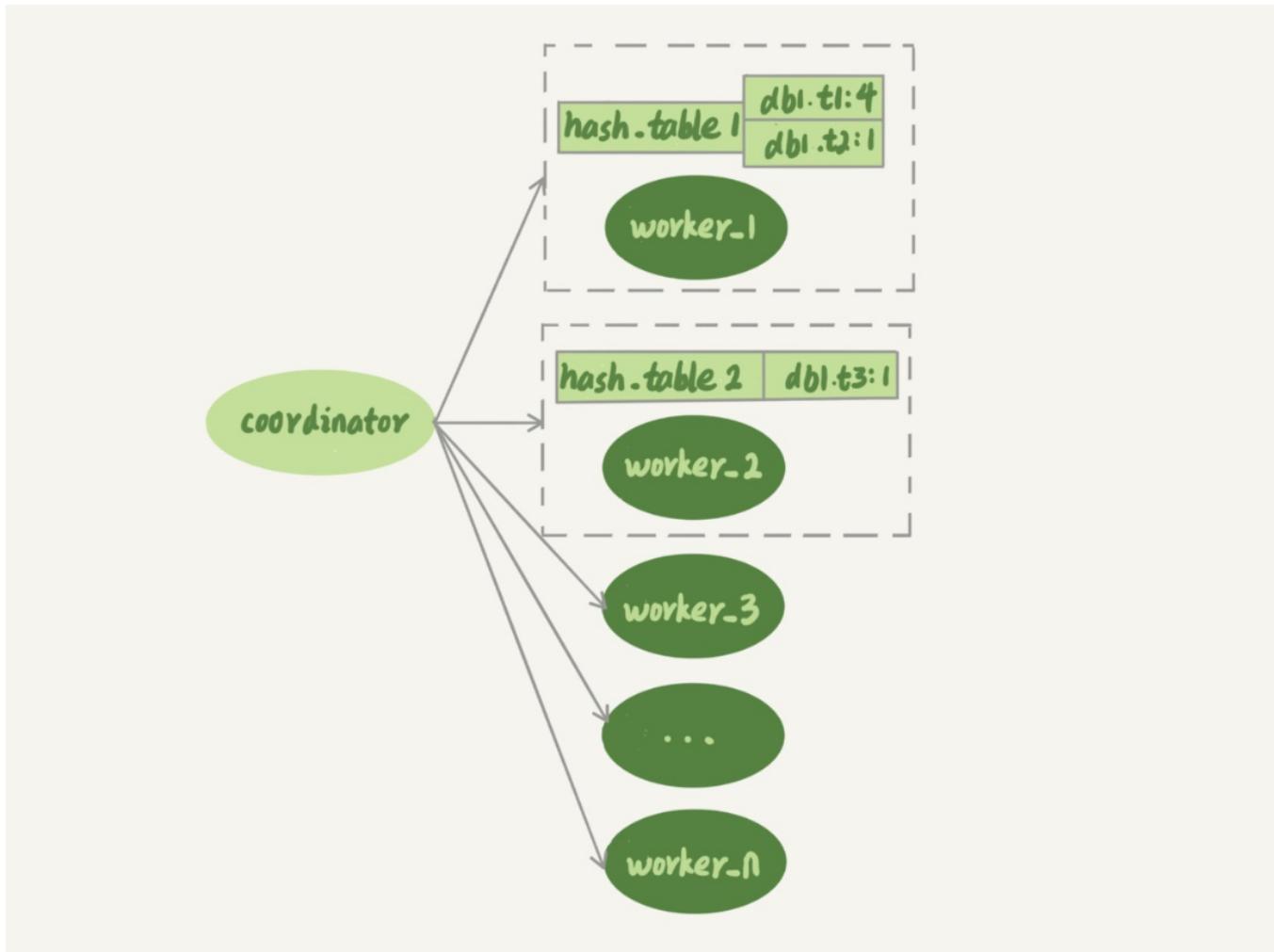


图3 按表并行复制程模型

可以看到，每个**worker**线程对应一个**hash**表，用于保存当前正在这个**worker**的“执行队列”里的事务所涉及的表。**hash**表的**key**是“库名.表名”，**value**是一个数字，表示队列中有多少个事务修改这个表。

在有事务分配给**worker**时，事务里面涉及的表会被加到对应的**hash**表中。**worker**执行完成后，这个表会被从**hash**表中去掉。

图3中，**hash_table_1**表示，现在**worker_1**的“待执行事务队列”里，有4个事务涉及到**db1.t1**表，有1个事务涉及到**db2.t2**表；**hash_table_2**表示，现在**worker_2**中有一个事务会更新到表**t3**的数据。

假设在图中的情况下，**coordinator**从中转日志中读入一个新事务T，这个事务修改的行涉及到表**t1**和**t3**。

现在我们用事务T的分配流程，来看一下分配规则。

1. 由于事务T中涉及修改表t1，而**worker_1**队列中有事务在修改表t1，事务T和队列中的某个事务要修改同一个表的数据，这种情况我们说事务T和**worker_1**是冲突的。

2. 按照这个逻辑，顺序判断事务T和每个worker队列的冲突关系，会发现事务T跟worker_2也冲突。
3. 事务T跟多于一个worker冲突，coordinator线程就进入等待。
4. 每个worker继续执行，同时修改hash_table。假设hash_table_2里面涉及到修改表t3的事务先执行完成，就会从hash_table_2中把db1.t3这一项去掉。
5. 这样coordinator会发现跟事务T冲突的worker只有worker_1了，因此就把它分配给worker_1。
6. coordinator继续读下一个中转日志，继续分配事务。

也就是说，每个事务在分发的时候，跟所有worker的冲突关系包括以下三种情况：

1. 如果跟所有worker都不冲突，coordinator线程就会把这个事务分配给最空闲的worker；
2. 如果跟多于一个worker冲突，coordinator线程就进入等待状态，直到和这个事务存在冲突关系的worker只剩下1个；
3. 如果只跟一个worker冲突，coordinator线程就会把这个事务分配给这个存在冲突关系的worker。

这个按表分发的方案，在多个表负载均匀的场景里应用效果很好。但是，如果碰到热点表，比如所有的更新事务都会涉及到某一个表的时候，所有事务都会被分配到同一个worker中，就变成单线程复制了。

按行分发策略

要解决热点表的并行复制问题，就需要一个按行并行复制的方案。按行复制的核心思路是：如果两个事务没有更新相同的行，它们在备库上可以并行执行。显然，这个模式要求binlog格式必须是row。

这时候，我们判断一个事务T和worker是否冲突，用的规则就不是“修改同一个表”，而是“修改同一行”。

按行复制和按表复制的数据结构差不多，也是为每个worker，分配一个hash表。只是要实现按行分发，这时候的key，就必须是“库名+表名+唯一键的值”。

但是，这个“唯一键”只有主键id还是不够的，我们还需要考虑下面这种场景，表t1中除了主键，还有唯一索引a：

```
CREATE TABLE `t1` (
```

```
    `id` int(11) NOT NULL,  
    `a` int(11) DEFAULT NULL,  
    `b` int(11) DEFAULT NULL,  
    PRIMARY KEY (`id`),  
    UNIQUE KEY `a` (`a`)  
) ENGINE=InnoDB;
```

```
insert into t1 values(1,1,1),(2,2,2),(3,3,3),(4,4,4),(5,5,5);
```

假设，接下来我们要在主库执行这两个事务：

session A	session B
update t1 set a=6 where id=1;	
	update t1 set a=1 where id=2;

图4 唯一键冲突示例

可以看到，这两个事务要更新的行的主键值不同，但是如果它们被分到不同的**worker**，就有可能**session B**的语句先执行。这时候**id=1**的行的**a**的值还是1，就会报唯一键冲突。

因此，基于行的策略，事务**hash**表中还需要考虑唯一键，即**key**应该是“库名+表名+索引**a**的名字+**a**的值”。

比如，在上面这个例子中，我要在表**t1**上执行**update t1 set a=1 where id=2**语句，在**binlog**里面记录了整行的数据修改前各个字段的值，和修改后各个字段的值。

因此，**coordinator**在解析这个语句的**binlog**的时候，这个事务的**hash**表就有三个项：

1. **key=hash_func(db1+t1+“PRIMARY”+2), value=2;** 这里**value=2**是因为修改前后的行**id**值不变，出现了两次。
2. **key=hash_func(db1+t1+“a”+2), value=1,** 表示会影响到这个表**a=2**的行。
3. **key=hash_func(db1+t1+“a”+1), value=1,** 表示会影响到这个表**a=1**的行。

可见，相比于按表并行分发策略，按行并行策略在决定线程分发的时候，需要消耗更多的计算资源。你可能也发现了，这两个方案其实都有一些约束条件：

1. 要能够从**binlog**里面解析出表名、主键值和唯一索引的值。也就是说，主库的**binlog**格式必

须是**row**;

2. 表必须有主键；
3. 不能有外键。表上如果有外键，级联更新的行不会记录在**binlog**中，这样冲突检测就不准确。

但，好在这三条约束规则，本来就是**DBA**之前要求业务开发人员必须遵守的线上使用规范，所以这两个并行复制策略在应用上也没有碰到什么麻烦。

对比按表分发和按行分发这两个方案的话，按行分发策略的并行度更高。不过，如果是要操作很多行的大事务的话，按行分发的策略有两个问题：

1. 耗费内存。比如一个语句要删除100万行数据，这时候**hash**表就要记录100万个项。
2. 耗费**CPU**。解析**binlog**，然后计算**hash**值，对于大事务，这个成本还是很高的。

所以，我在实现这个策略的时候会设置一个阈值，单个事务如果超过设置的行数阈值（比如，如果单个事务更新的行数超过10万行），就暂时退化为单线程模式，退化过程的逻辑大概是这样的：

1. **coordinator**暂时先**hold**住这个事务；
2. 等待所有**worker**都执行完成，变成空队列；
3. **coordinator**直接执行这个事务；
4. 恢复并行模式。

读到这里，你可能会感到奇怪，这两个策略又没有被合到官方，我为什么要介绍这么详细呢？其实，介绍这两个策略的目的是抛砖引玉，方便你理解后面要介绍的社区版本策略。

MySQL 5.6版本的并行复制策略

官方MySQL5.6版本，支持了并行复制，只是支持的粒度是按库并行。理解了上面介绍的按表分发策略和按行分发策略，你就理解了，用于决定分发策略的**hash**表里，**key**就是数据库名。

这个策略的并行效果，取决于压力模型。如果在主库上有多个**DB**，并且各个**DB**的压力均衡，使用这个策略的效果会很好。

相比于按表和按行分发，这个策略有两个优势：

1. 构造**hash**值的时候很快，只需要库名；而且一个实例上**DB**数也不会很多，不会出现需要构造100万个项这种情况。

2. 不要求binlog的格式。因为statement格式的binlog也可以很容易拿到库名。

但是，如果你的主库上的表都放在同一个DB里面，这个策略就没有效果了；或者如果不同DB的热点不同，比如一个是业务逻辑库，一个是系统配置库，那也起不到并行的效果。

理论上你可以创建不同的DB，把相同热度的表均匀分到这些不同的DB中，强行使用这个策略。不过据我所知，由于需要特地移动数据，这个策略用得并不多。

MariaDB的并行复制策略

在[第23篇文章](#)中，我给你介绍了redo log组提交(group commit)优化，而MariaDB的并行复制策略利用的就是这个特性：

1. 能够在同一组里提交的事务，一定不会修改同一行；
2. 主库上可以并行执行的事务，备库上也一定是可以并行执行的。

在实现上，MariaDB是这么做的：

1. 在一组里面一起提交的事务，有一个相同的commit_id，下一组就是commit_id+1；
2. commit_id直接写到binlog里面；
3. 传到备库应用的时候，相同commit_id的事务分发到多个worker执行；
4. 这一组全部执行完成后，coordinator再去取下一批。

当时，这个策略出来的时候是相当惊艳的。因为，之前业界的思路都是在“分析binlog，并拆分到worker”上。而MariaDB的这个策略，目标是“模拟主库的并行模式”。

但是，这个策略有一个问题，它并没有实现“真正的模拟主库并发度”这个目标。在主库上，一组事务在commit的时候，下一组事务是同时处于“执行中”状态的。

如图5所示，假设了三组事务在主库的执行情况，你可以看到在trx1、trx2和trx3提交的时候，trx4、trx5和trx6是在执行的。这样，在第一组事务提交完成的时候，下一组事务很快就会进入commit状态。

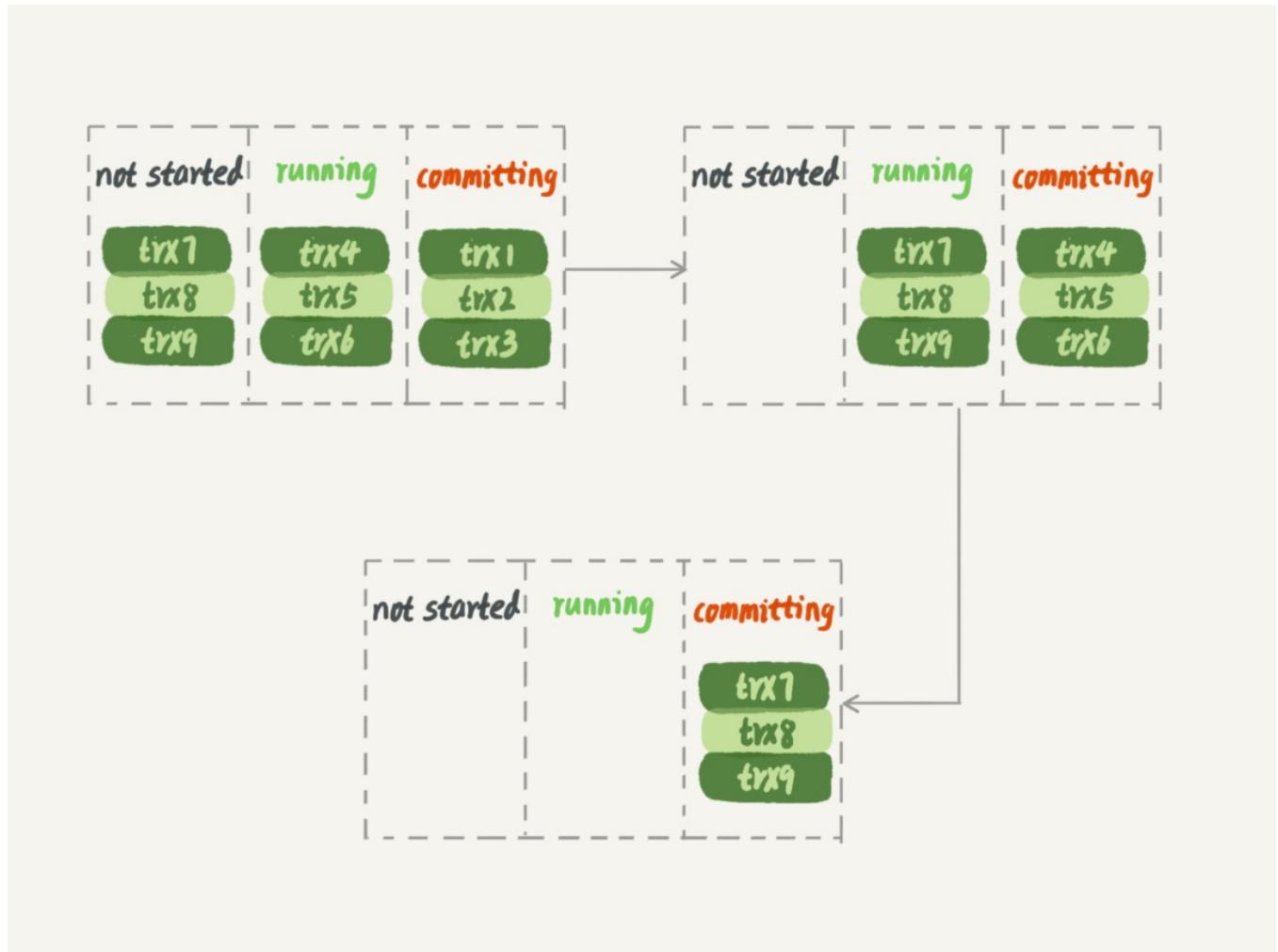


图5 主库并行事务

而按照MariaDB的并行复制策略，备库上的执行效果如图6所示。

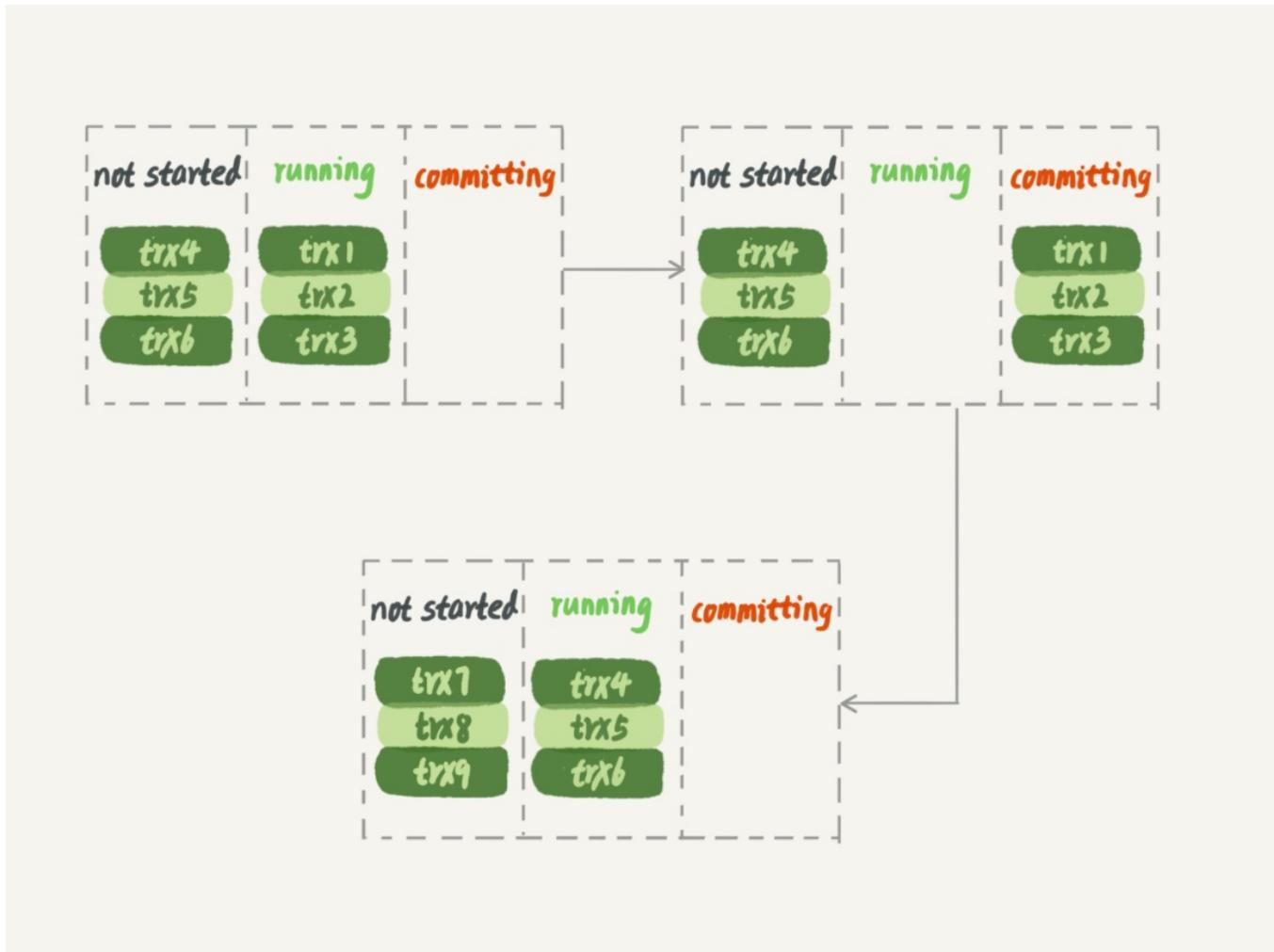


图6 MariaDB 并行复制，备库并行效果

可以看到，在备库上执行的时候，要等第一组事务完全执行完成后，第二组事务才能开始执行，这样系统的吞吐量就不够。

另外，这个方案很容易被大事务拖后腿。假设trx2是一个超大事务，那么在备库应用的时候，trx1和trx3执行完成后，就只能等trx2完全执行完成，下一组才能开始执行。这段时间，只有一个worker线程在工作，是对资源的浪费。

不过即使如此，这个策略仍然是一个很漂亮的创新。因为，它对原系统的改造非常少，实现也很优雅。

MySQL 5.7的并行复制策略

在MariaDB并行复制实现之后，官方的MySQL5.7版本也提供了类似的功能，由参数`slave_parallel_type`来控制并行复制策略：

1. 配置为`DATABASE`，表示使用MySQL 5.6版本的按库并行策略；
2. 配置为`LOGICAL_CLOCK`，表示的就是类似MariaDB的策略。不过，MySQL 5.7这个策略，针对并行度做了优化。这个优化的思路也很有趣儿。

你可以先考虑这样一个问题：同时处于“执行状态”的所有事务，是不是可以并行？

答案是，不能。

因为，这里面可能有由于锁冲突而处于锁等待状态的事务。如果这些事务在备库上被分配到不同的**worker**，就会出现备库跟主库不一致的情况。

而上面提到的**MariaDB**这个策略的核心，是“所有处于**commit**”状态的事务可以并行。事务处于**commit**状态，表示已经通过了锁冲突的检验了。

这时候，你可以再回顾一下两阶段提交，我把前面[第23篇文章](#)中介绍过的两阶段提交过程图贴过来。

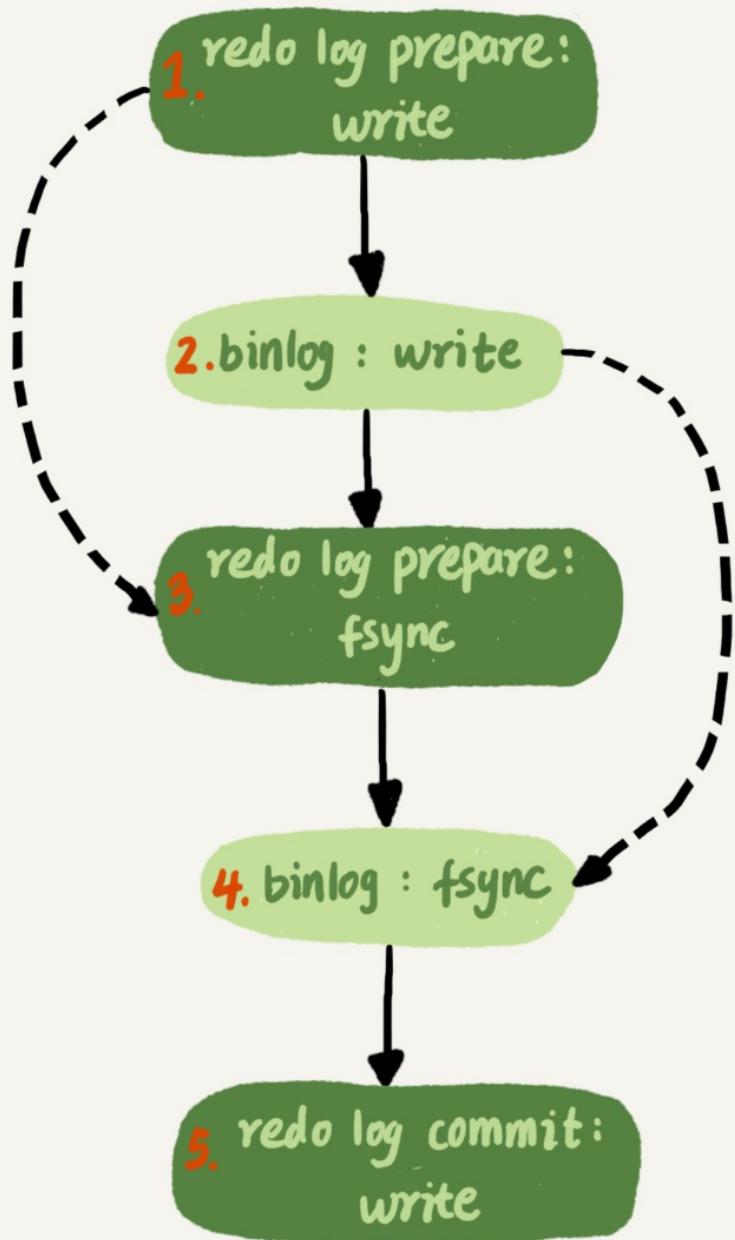


图7 两阶段提交细化过程图

其实，不用等到commit阶段，只要能够到达redo log prepare阶段，就表示事务已经通过锁冲突的检验了。

因此，MySQL 5.7并行复制策略的思想是：

1. 同时处于**prepare**状态的事务，在备库执行时是可以并行的；
2. 处于**prepare**状态的事务，与处于**commit**状态的事务之间，在备库执行时也是可以并行的。

我在第23篇文章，讲**binlog**的组提交的时候，介绍过两个参数：

1. **binlog_group_commit_sync_delay**参数，表示延迟多少微秒后才调用**fsync**；
2. **binlog_group_commit_sync_no_delay_count**参数，表示累积多少次以后才调用**fsync**。

这两个参数是用于故意拉长**binlog**从**write**到**fsync**的时间，以此减少**binlog**的写盘次数。在MySQL 5.7的并行复制策略里，它们可以用来制造更多的“同时处于**prepare**阶段的事务”。这样就增加了备库复制的并行度。

也就是说，这两个参数，既可以“故意”让主库提交得慢些，又可以让备库执行得快些。在MySQL 5.7处理备库延迟的时候，可以考虑调整这两个参数值，来达到提升备库复制并发度的目的。

MySQL 5.7.22的并行复制策略

在2018年4月份发布的MySQL 5.7.22版本里，MySQL增加了一个新的并行复制策略，基于**WRITESET**的并行复制。

相应地，新增了一个参数**binlog-transaction-dependency-tracking**，用来控制是否启用这个新策略。这个参数的可选值有以下三种。

1. **COMMIT_ORDER**，表示的就是前面介绍的，根据同时进入**prepare**和**commit**来判断是否可以并行的策略。
2. **WRITESET**，表示的是对于事务涉及更新的每一行，计算出这一行的**hash**值，组成集合**writeset**。如果两个事务没有操作相同的行，也就是说它们的**writeset**没有交集，就可以并行。
3. **WRITESET_SESSION**，是在**WRITESET**的基础上多了一个约束，即在主库上同一个线程先后执行的两个事务，在备库执行的时候，要保证相同的先后顺序。

当然为了唯一标识，这个**hash**值是通过“库名+表名+索引名+值”计算出来的。如果一个表上除了有主键索引外，还有其他唯一索引，那么对于每个唯一索引，**insert**语句对应的**writeset**就要多增加一个**hash**值。

你可能看出来了，这跟我们前面介绍的基于MySQL 5.5版本的按行分发的策略是差不多的。不过，MySQL官方的这个实现还是有很大的优势：

1. **writeset**是在主库生成后直接写入到**binlog**里面的，这样在备库执行的时候，不需要解析**binlog**内容（**event**里的行数据），节省了很多计算量；
2. 不需要把整个事务的**binlog**都扫一遍才能决定分发到哪个**worker**，更省内存；
3. 由于备库的分发策略不依赖于**binlog**内容，所以**binlog**是**statement**格式也是可以的。

因此，MySQL 5.7.22的并行复制策略在通用性上还是有保证的。

当然，对于“表上没主键”和“外键约束”的场景，**WRITESET**策略也是没法并行的，也会暂时退化为单线程模型。

小结

在今天这篇文章中，我和你介绍了MySQL的各种多线程复制策略。

为什么要有多线程复制呢？这是因为单线程复制的能力全面低于多线程复制，对于更新压力较大的主库，备库是可能一直追不上主库的。从现象上看就是，备库上**seconds_behind_master**的值越来越大。

在介绍完每个并行复制策略后，我还和你分享了不同策略的优缺点：

- 如果你是**DBA**，就需要根据不同的业务场景，选择不同的策略；
- 如果是你业务开发人员，也希望你能从中获取灵感用到平时的开发工作中。

从这些分析中，你也会发现大事务不仅会影响到主库，也是造成备库复制延迟的主要原因之一。因此，在平时的开发工作中，我建议你尽量减少大事务操作，把大事务拆成小事务。

官方MySQL 5.7版本新增的备库并行策略，修改了**binlog**的内容，也就是说**binlog**协议并不是向上兼容的，在主备切换、版本升级的时候需要把这个因素也考虑进去。

最后，我给你留下一个思考题吧。

假设一个MySQL 5.7.22版本的主库，单线程插入了很多数据，过了3个小时后，我们要给这个主库搭建一个相同版本的备库。

这时候，你为了更快地让备库追上主库，要开并行复制。在**binlog-transaction-dependency-tracking**参数的**COMMIT_ORDER**、**WRITESET**和**WRITE_SESSION**这三个取值中，你会选择哪一个呢？

你选择的原因是什么？如果设置另外两个参数，你认为会出现什么现象呢？

你可以把你的答案和分析写在评论区，我会在下一篇文章跟你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，什么情况下，备库的主备延迟会表现为一个45度的线段？评论区有不少同学的回复都说到重点：备库的同步在这段时间完全被堵住了。

产生这种现象典型的场景主要包括两种：

- 一种是大事务（包括大表**DDL**、一个事务操作很多行）；
- 还有一种情况比较隐蔽，就是备库起了一个长事务，比如

```
begin;  
select * from t limit 1;
```

然后就不动了。

这时候主库对表t做了一个加字段操作，即使这个表很小，这个**DDL**在备库应用的时候也会被堵住，也不能看到这个现象。

评论区还有同学说是是不是主库多线程、从库单线程，备库跟不上主库的更新节奏导致的？今天这篇文章，我们刚好讲的是并行复制。所以，你知道了，这种情况会导致主备延迟，但不会表现为这种标准的呈45度的直线。

评论区留言点赞板：

@易翔、@万勇、@老杨同志 等同学的回复都提到了我们上面说的场景；

@Max 同学提了一个很不错的问题。主备关系里面，备库主动连接，之后的**binlog**发送是主库主动推送的。之所以这么设计也是为了效率和实时性考虑，毕竟靠备库轮询，会有时间差。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「」请朋友读」，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



老杨同志

5

尝试回答 慧鑫coming 的问题。

老师图片的步骤有下面5步

1 redo log prepare write

2 binlog write

3 redo log prepare fsync

4 binlog fsync

5 redo log commit write

1)如果更新通一条记录是有锁的，只能一个事务执行，其他事务等待锁。

2)第4步的时候会因为下面两个参数，等其他没有锁冲突的事务，一起刷盘，此时一起执行的事务拥有相同的**commit_id**

`binlog_group_commit_sync_delay`

`binlog_group_commit_sync_no_delay_count`

3)执行步骤5后，释放锁，等待锁的事务开始执行。

所以对同一行更新的事务，不可能拥有相同的**commit_id**

2019-01-11

作者回复

[], 你比我回复得详细，顶起

2019-01-11



长杰

冂 2

举个例子，一个事务更新了表 **t1** 和表 **t2** 中的各一行，如果这两条更新语句被分到不同 **worker** 的话，虽然最终的结果是主备一致的，但如果表 **t1** 执行完成的瞬间，备库上有一个查询，就会看到这个事务“更新了一半的结果”，破坏了事务逻辑的原子性。

老师这块不太明白，备库有查询会看到更新了一半的结果，**t1**的**worker**执行完了更新会**commit**吗？如果不**commit**，备库查询应该看不到吧？如果**commit**，就破坏了事物的原子性，肯定是有问题的。

2019-01-11

| 作者回复

应该是说，它迟早要**commit**，但是两个**worker**是两个线程，没办法约好“同时提交”，这样就有可能出现一个先提交一个后提交。

这两个提交之间的时间差，就能被用户看到“一半事务”，好问题

2019-01-11



jike

冂 1

老师您好，开启并行复制后，事务是按照组来提交的，从库也是根据**commit_id**来回放，如果从库也开启**binlog**的话，那是不是存在主从的**binlog event**写入顺序不一致的情况呢？

2019-01-15

| 作者回复

是有可能**binlog event**写入顺序不同的，好问题

2019-01-15



HuaMax

冂 7

课后题。关键点在于主库单线程，针对三种不同的策略，**COMMIT_ORDER**: 没有同时到达**redo log**的**prepare**状态的事务，备库退化为单线程；**WRITESET**: 通过对比更新的事务是否存在冲突的行，可以并发执行；**WRITE_SESSION**: 在**WRITESET**的基础上增加了线程的约束，则退化为单线程。综上，应选择**WRITESET**策略

2019-01-12

| 作者回复

准确[]

2019-01-12



慧鑫coming

冂 2

老师，有个问题，**mariadb**的并行策略，当同一组中有3个事务，它们都对同一行同一字段值进行更改，而它们的**commit_id**相同，可以在从库并行执行，那么3者的先后顺序是怎么保证不影响该行该字段的最终结果与主库一致？

2019-01-11

| 作者回复

好问题

不过这个是不可能的哈，对同一行的修改，第一个拿到行锁的事务还没提交前，另外两个会被行锁堵住的，这两个进入不了commit状态。所以这三个的commit_id不会相同的。

2019-01-11



IceGeek17

1

好文，总结对比不同的并行策略，讲的深入浅出，看完豁然开朗。有看源代码的冲动。

2019-01-24

| 作者回复

看完分享你的心得哈。

2019-01-24



每天晒白牙

1

我是做java的，看老师的这个专栏，确实挺吃力的，老师专栏的干货太多了，下面的留言也是相当有水平，质量都很高，互动也好，应该是好多DBA吧，做java的我，看的头大

2019-01-13

| 作者回复

这几篇偏深，但确实是大家在使用的时候需要了解的，

到30篇后面的文章会偏应用哈

2019-01-13



某、人

1

总结下多线程复制的流程,有不对之处请老师指出:

双1,配置为logical_clock,假设有三个事务并发执行也已经执行完成(都处于prepare阶段)

1.三个事务把redo log从redo log buffer写到fs page cache中

2.把binlog_cache flush到binlog文件中,最先进入flush队列的为leader,

其它两个事务为follower.把组员编号以及组的编号写进binlog文件中(三个事务为同一组).

3.三个事务的redo log做fsync,binlog做fsync.

4.dump线程从binlog文件里把binlog event发送给从库

5.I/O线程接收到binlog event,写到relay log中

6.sql thread读取relay log,判断出这三个事务是处于同一个组,

则把这三个事务的event打包发送给三个空闲的worker线程(如果有)并执行。

配置为writeset的多线程复制流程:

1.三个事务把redo log从redo log buffer写到fs page cache中

2.把binlog_cache flush到binlog文件中,根据表名、主键和唯一键(如果有)生成hash值(writeset),

保存到hash表中

判断这三个事务的writeset是否有冲突,如果没有冲突,则视为同组,如果有冲突,则视为不同组.

并把组员编号以及组的编号写进binlog文件中

(不过一个组的事务个数也不是无限大,由参数binlog_transaction_dependency_history_size决定组内最多事务数)

3.然后做redo log和binlog的fsync

4.dump线程从binlog文件里把binlog event发送给从库

5.I/O线程接收到binlog event,写到relay log中

6.sql thread读取relay log,如果是同一个组的事务,则把事务分配到不同的worker线程去应用relay

log.

不同组的事务,需要等到上一个组的事务全部执行完成,才能分配worker线程应用relay log.

老师我有几个问题想请教下:

1.在备库是单线程下,second_behind_master是通过计算T3-T1得到,

在多线程的情况下,是怎么计算出second_behind_master的值?用的是哪一个事务的时间戳?

2.多线程复制下,如果从库宕机了,是不是从库有一个记录表记录那些事务已经应用完成,恢复的时候,只需要恢复未应用的事务.

3.binlog延迟sync的两个参数,是延迟已经flush未sync时间。意思是让事务组占用flush时间更长,之后的事务有更多的时间,从binlog cache进入到flush队列,使得组员变多,起到从库并发的目的因为我理解的是加入到组是在binlog cache flush到binlog文件之前做的,如果此时有事务正在flush,

未sync,则后面的事务必须等待。不知道理解得对不对

2019-01-13

| 作者回复

上面的描述部分, writeset的多线程复制流程里面, 这段需要修改下:

『2.把binlog_cache flush到binlog文件中,根据表名、主键和唯一键(如果有)生成hash值(writeset),保存到hash表中

【判断这三个事务的writeset是否有冲突,如果没有冲突,则视为同组,如果有冲突,则视为不同组.并把组员编号以及组的编号写进binlog文件中】』

上面中括号这段要去掉,

判断writeset之间是否可以并行这个逻辑, 是在备库的coordinator线程做的。

1. 在多线程并发的时候, Seconds_behind_master很不准, 后面会介绍别的判断方法;

2. 是的,备库有记录, 就是show slave status 里面的Relay_Log_File 和 Relay_Log_Pos 这两个值表示的, 好问题

3. ”加入到组是在binlog cache flush到binlog文件之前做的,如果此时有事务正在flush,未sync,则后面的事务必须等待“这句话是对的, 但是我没看出这个跟前面提的两个延迟参数作用的关系^

^

2019-01-13



观弈道人

1

丁老师你好, 问个题外问题, mysql已经通过gap锁解决了在rr级别下的幻读问题, 那么serializable隔离级别目前还有什么用途, 一般文章上说的, serializable主要是为了解决幻读, 谢谢回答。

2019-01-12

| 作者回复

Serializable隔离级别确实用得很少（我没有见过在生产上使用的哈）

2019-01-12



J!

0

同时处于 **prepare** 状态的事务，在备库执行时是可以并行复制的，是这个**prepare** 就可以生成了改组的**committed Id**吗

极客时间版权所有: <https://time.geekbang.org/column/article/77083>

2019-02-01

| 作者回复

进入**prepare** 的时候就给这个事务分配 **commitid**，这个**commitid**就是当前系统最大的一个**commitid**

2019-02-02



J!

0

5.7 版本的基于组提交的并行复制。**last_commitid** 是在什么时候生成的？

2019-02-01

| 作者回复

事务提交的时候

2019-02-02



alias cd=rm -rf

0

老师您好：

思考题答案的猜测：建议采用 **WRITESET**。

WRITESET_SESSION: 因为主库是单线程插入，如果采用**WRITESET_SESSION**，那么会退化成单线程主从复制。

COMMIT_ORDER: 因为是追历史数据，所以会退化成单线程。

2019-02-01

| 作者回复

对的，

2019-02-02



时隐时现

0

Furthermore, given that changes are propagated and applied in row-based format, this means that they are received in an optimized and compact format, and likely reducing the number of IO operations required when compared to the originating member.

这个是官档上对MGR的一段解读，我的疑问是：

为何**row-base replication**在从库回放时会节省大量IO？

候选答案：

1、省去了**sql**解析，直接调用**do_command**

2、？？

可是**row**复制有其他可能存在的劣势，比如单个大**dml**会被解析成多个**dml_event**进行重放，万

一该表没有主键或唯一索引，只能采用二级索引或者全表扫描(开启hash_scan也可以)，所以，官档上直接说会减少大量IO是不是有点太武断了

2019-01-31

作者回复

这个描述应该是主要考虑在有主键的时候，可以通过row里面的信息取出主键直接定位记录。

你说的这些其实劣势确实也是存在的『

2019-01-31



牛牛

0

老师、请教两个问题~

1. 我在job里按主键删除线上表数据的时候、造成了主从延迟、`delete from table where id in...`

`id`是主键、每次`delete 300条`、`sleep 500ms`、这种延迟可能是什么造成的呢？`300条`应该不算大事务？还是说快速的数据删除导致了索引重建？

2. 如果一个表快速往里写数据、每次`300条`、`sleep 1s`、这个库上的读取会慢吗？

多谢老师~

2019-01-27

作者回复

1. `delete 300条`、`sleep 500ms`已经是很克制的操作了，单线程吗？如果还是单线程，那延迟应该不是这个操作导致的

2. 这都是很小的压力，不会读取慢才对

2019-02-01



Leon

0

老师，`semisync`啥时候讲下，昨天面试被问到一脸懵逼

2019-01-22

作者回复

`semi-sync`在第28篇会提到，但是也不是大篇幅介绍

后面可能也不会大篇幅专门介绍了，你说下你的问题哈。

2019-01-22



Mr.Strive.Z.H.L

0

老师您好：

关于`COMMIT_ORDER`的并行复制方案，从库根据`commit_id`来判断“处于`prepare`和`commit`状态的事务”。这里我有个很大的疑惑：`commit_id`是什么时候加入到`binlog`的，又是在什么时候递增的？？

(

对于我这个问题的进一步解释：

既然commit_id是要被写入到binlog的，那么commit_id毫无疑问就是在write binlog阶段写入的。
。

我们知道redolog是组提交的，如果只是按照redolog的组提交方式生成commit_id，那么这个commit_id包含的并行事务数量并不够多！因为在binlog write阶段，又有事务进入到redolog prepare阶段，他们之间的commit_id是不一样的，但是他们是可以并行的。

所以commit_id什么时候递增？这个是非常关键的，我也很疑惑，commit_id到底是根据什么条件递增的？？

)

2019-01-17

| 作者回复

可以这么理解，每个事务都有两个数字表示它在执行提交阶段的时间范围，构成区间(c1, c2)。如果两个事务的区间有交集，就是可以并行的。

这里c1是事务启动的时候，当前系统里最大的commit_id；

一个事务提交的时候，commit_id+1.

2019-01-17



Mr.Strive.Z.H.L

0

老师您好：

今天的内容中写到：“外键约束”会导致并行复制退化为单线程。

这个地方我就突然联想到，在业务中，类似于“外键”这种关系是一定存在的。但是一般在设计表的时候，比如：表A的某个唯一键是表B的外键。并不会真正“显示”的在数据库表中创建外键关系。（查询的时候，查询出A的这个唯一键，然后再根据这个唯一键查询表B的数据，并不会有真正的外键关系，一次性查出所有关联数据）

这是为什么呢？

2019-01-17

| 作者回复

我也建议尽量少使用外键，我自己理解的几个原因吧

1. 这个关系应该维护在开发系统的逻辑中，放在数据库里面，比较隐蔽，容易忘记
2. 外键约束可能会导致有些更新失败
3. 外键约束（尤其是级联更新）容易出现非预期的结果

2019-01-17



玳星东

0

老师好，如何将大事务拆成小事务

2019-01-16

| 作者回复

这个是要结合业务的，比如要删除100万行，改成100个事务，每个事务删除1万行，这样的

2019-01-16



道

0

老师，这段不太理解：“举个例子，一个事务更新了表 t1 和表 t2 中的各一行，如果这两条更新

语句被分到不同 **worker** 的话，虽然最终的结果是主备一致的，但如果表 **t1** 执行完成的瞬间，备库上有一个查询，就会看到这个事务“更新了一半的结果”，破坏了事务逻辑的原子性。”备库上的查询属于另外一个事务，按照可重复读隔离级别，这个查询不应该看到另外一个事务“更新了一半的结果”啊。即便是这两条更新语句被分到不同 **worker**，也应该保证事务的原子性啊，难道是技术上有困难吗？

2019-01-16

作者回复

因为这两个**worker**没办法“约好一起提交”，这个是属于两个线程了

2019-01-16



crazyone

0

“不用等到 **commit** 阶段，只要能够到达 **redo log prepare** 阶段，就表示事务已经通过锁冲突的检验了。”这句话不怎么理解。事务获取锁是在执行到对应的语句才做检查的，**redo log** 在事务当中，应该也是一条条操作语句写的吧？难道写完了，才会进入到**prepare**阶段？这个**prepare**阶段是指事务已经完全扫描执行完所有事务操作，准备写入到**redo log**文件的阶段？

2019-01-15

作者回复

就是两阶段提交里的，写**redo**的第一阶段

2019-01-16

27 | 主库出问题了，从库怎么办？

2019-01-14 林晓斌



在前面的第[24](#)、[25](#)和[26](#)篇文章中，我和你介绍了MySQL主备复制的基础结构，但这些都是一主一备的结构。

大多数的互联网应用场景都是读多写少，因此你负责的业务，在发展过程中很可能先会遇到读性能的问题。而在数据库层解决读性能问题，就要涉及到接下来两篇文章要讨论的架构：一主多从。

今天这篇文章，我们就先聊聊一主多从的切换正确性。然后，我们在下一篇文章中再聊聊解决一主多从的查询逻辑正确性的方法。

如图1所示，就是一个基本的一主多从结构。

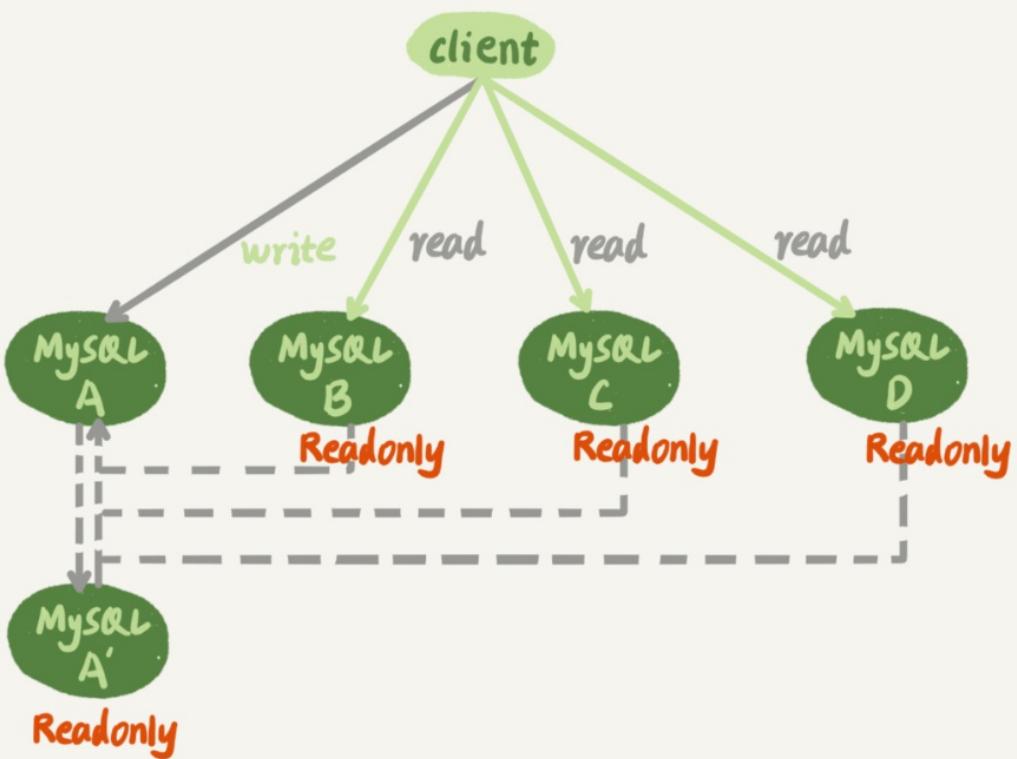


图1 一主多从基本结构

图中，虚线箭头表示的是主备关系，也就是A和A'互为主备，从库B、C、D指向的是主库A。一主多从的设置，一般用于读写分离，主库负责所有的写入和一部分读，其他的读请求则由从库分担。

今天我们要讨论的就是，在一主多从架构下，主库故障后的主备切换问题。

如图2所示，就是主库发生故障，主备切换后的结果。

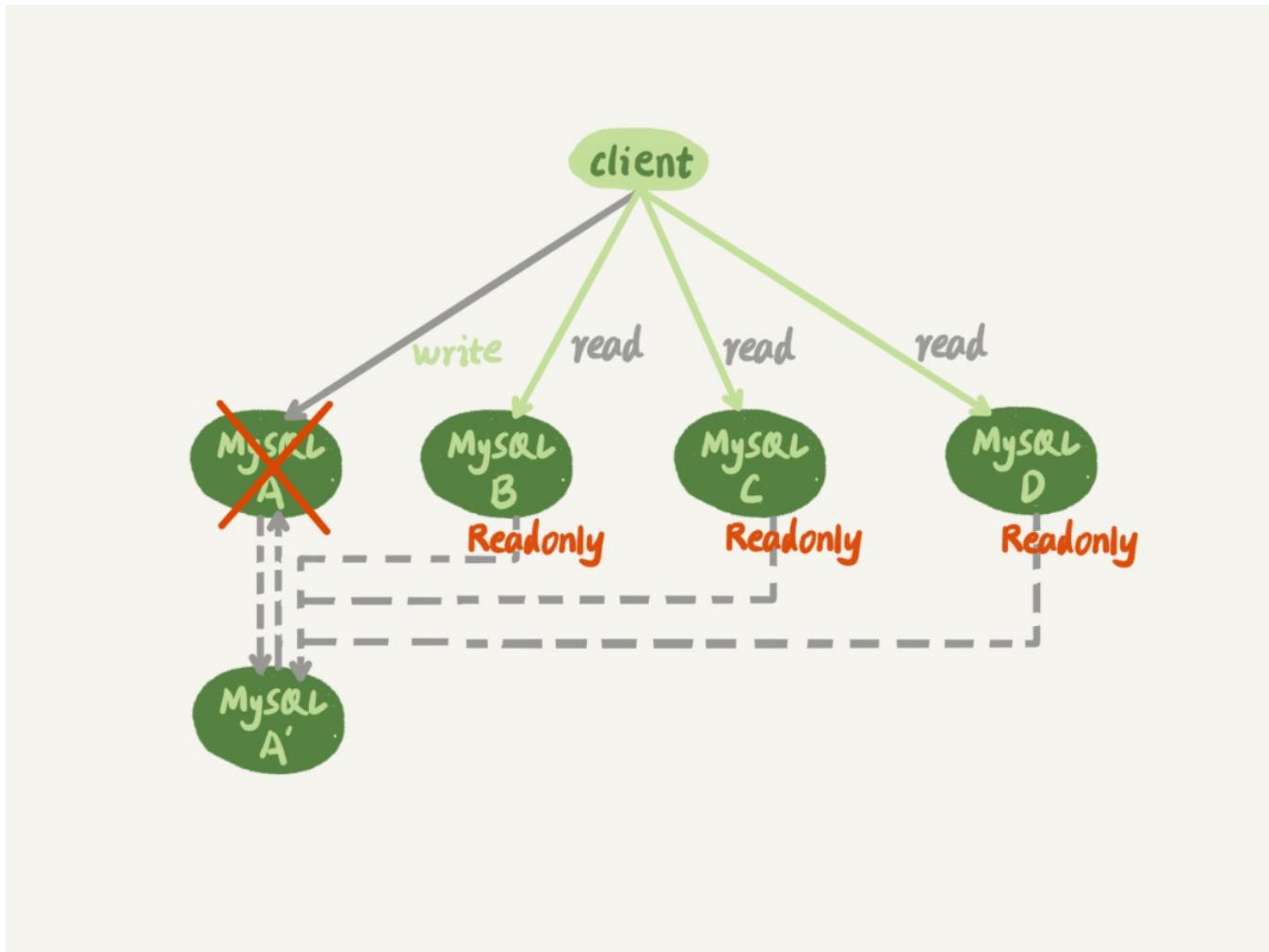


图2 一主多从基本结构—主备切换

相比于一主一备的切换流程，一主多从结构在切换完成后，**A'**会成为新的主库，从库**B、C、D**也要改接到**A'**。正是由于多了从库**B、C、D**重新指向的这个过程，所以主备切换的复杂性也相应增加了。

接下来，我们再一起看看一个切换系统会怎么完成一主多从的主备切换过程。

基于位点的主备切换

这里，我们需要先来回顾一个知识点。

当我们把节点**B**设置成节点**A'**的从库的时候，需要执行一条**change master**命令：

```
CHANGE MASTER TO
MASTER_HOST=$host_name
MASTER_PORT=$port
MASTER_USER=$user_name
MASTER_PASSWORD=$password
MASTER_LOG_FILE=$master_log_name
MASTER_LOG_POS=$master_log_pos
```

这条命令有这么6个参数：

- **MASTER_HOST**、**MASTER_PORT**、**MASTER_USER**和**MASTER_PASSWORD**四个参数，分别代表了主库A'的IP、端口、用户名和密码。
- 最后两个参数**MASTER_LOG_FILE**和**MASTER_LOG_POS**表示，要从主库的**master_log_name**文件的**master_log_pos**这个位置的日志继续同步。而这个位置就是我们所说的同步位点，也就是主库对应的文件名和日志偏移量。

那么，这里就有一个问题了，节点B要设置成A'的从库，就要执行**change master**命令，就不可避免地要设置位点的这两个参数，但是这两个参数到底应该怎么设置呢？

原来节点B是A的从库，本地记录的也是A的位点。但是相同的日志，A的位点和A'的位点是不同的。因此，从库B要切换的时候，就需要先经过“找同步位点”这个逻辑。

这个位点很难精确取到，只能取一个大概位置。为什么这么说呢？

我来和你分析一下看看这个位点一般是怎么获取到的，你就清楚其中不精确的原因了。

考虑到切换过程中不能丢数据，所以我们找位点的时候，总是要找一个“稍微往前”的，然后再通过判断跳过那些在从库B上已经执行过的事务。

一种取同步位点的方法是这样的：

1. 等待新主库A'把中转日志（**relay log**）全部同步完成；
2. 在A'上执行**show master status**命令，得到当前A'上最新的File 和 Position；
3. 取原主库A故障的时刻T；
4. 用**mysqlbinlog**工具解析A'的File，得到T时刻的位点。

```
mysqlbinlog File --stop-datetime=T --start-datetime=T
```

```
/*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=1*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#190106 17:52:40 server id 1 end_log_pos 123 CRC32 0x5f3391fc  Start: binlog v 4, server v 5.7.21-log created 190106 17:52:40 at startup
# Warning: this binlog is either in use or was not closed properly.
```

图3 mysqlbinlog 部分输出结果

图中，**end_log_pos**后面的值“123”，表示的就是A'这个实例，在T时刻写入新的**binlog**的位置。然后，我们就可以把123这个值作为**\$master_log_pos**，用在节点B的**change master**命令里。

当然这个值并不精确。为什么呢？

你可以设想有这么一种情况，假设在T这个时刻，主库A已经执行完成了一个insert语句插入了一行数据R，并且已经将binlog传给了A'和B，然后在传完的瞬间主库A的主机就掉电了。

那么，这时候系统的状态是这样的：

1. 在从库B上，由于同步了binlog，R这一行已经存在；
2. 在新主库A'上，R这一行也已经存在，日志是写在123这个位置之后的；
3. 我们在从库B上执行change master命令，指向A'的File文件的123位置，就会把插入R这一行数据的binlog又同步到从库B去执行。

这时候，从库B的同步线程就会报告 Duplicate entry 'id_of_R' for key 'PRIMARY' 错误，提示出现了主键冲突，然后停止同步。

所以，通常情况下，我们在切换任务的时候，要先主动跳过这些错误，有两种常用的方法。

一种做法是，主动跳过一个事务。跳过命令的写法是：

```
set global sql_slave_skip_counter=1;  
start slave;
```

因为切换过程中，可能会不止重复执行一个事务，所以我们需要在从库B刚开始接到新主库A'时，持续观察，每次碰到这些错误就停下来，执行一次跳过命令，直到不再出现停下来的情况，以此来跳过可能涉及的所有事务。

另外一种方式是，通过设置slave_skip_errors参数，直接设置跳过指定的错误。

在执行主备切换时，有这么两类错误，是经常会遇到的：

- 1062错误是插入数据时唯一键冲突；
- 1032错误是删除数据时找不到行。

因此，我们可以把slave_skip_errors设置为“1032,1062”，这样中间碰到这两个错误时就直接跳过。

这里需要注意的是，这种直接跳过指定错误的方法，针对的是主备切换时，由于找不到精确的同步位点，所以只能采用这种方法来创建从库和新主库的主备关系。

这个背景是，我们很清楚在主备切换过程中，直接跳过1032和1062这两类错误是无损的，所以

才可以这么设置`slave_skip_errors`参数。等到主备间的同步关系建立完成，并稳定执行一段时间之后，我们还需要把这个参数设置为空，以免之后真的出现了主从数据不一致，也跳过了。

GTID

通过`sql_slave_skip_counter`跳过事务和通过`slave_skip_errors`忽略错误的方法，虽然都最终可以建立从库B和新主库A'的主备关系，但这两种操作都很复杂，而且容易出错。所以，MySQL 5.6版本引入了**GTID**，彻底解决了这个困难。

那么，**GTID**到底是什么意思，又是如何解决找同步位点这个问题呢？现在，我就和你简单介绍一下。

GTID的全称是**Global Transaction Identifier**，也就是全局事务ID，是一个事务在提交的时候生成的，是这个事务的唯一标识。它由两部分组成，格式是：

```
GTID=server_uuid:gno
```

其中：

- `server_uuid`是一个实例第一次启动时自动生成的，是一个全局唯一的值；
- `gno`是一个整数，初始值是1，每次提交事务的时候分配给这个事务，并加1。

这里我需要和你说明一下，在MySQL的官方文档里，**GTID**格式是这么定义的：

```
GTID=source_id:transaction_id
```

这里的`source_id`就是`server_uuid`；而后面的这个`transaction_id`，我觉得容易造成误导，所以我改成了`gno`。为什么说使用`transaction_id`容易造成误解呢？

因为，在MySQL里面我们说`transaction_id`就是指事务id，事务id是在事务执行过程中分配的，如果这个事务回滚了，事务id也会递增，而`gno`是在事务提交的时候才会分配。

从效果上看，**GTID**往往是连续的，因此我们用`gno`来表示更容易理解。

GTID模式的启动也很简单，我们只需要在启动一个MySQL实例的时候，加上参数`gtid_mode=on`和`enforce_gtid_consistency=on`就可以了。

在**GTID**模式下，每个事务都会跟一个**GTID**一一对应。这个**GTID**有两种生成方式，而使用哪种方式取决于**session**变量`gtid_next`的值。

1. 如果`gtid_next=automatic`，代表使用默认值。这时，MySQL就会把`server_uuid:gno`分配给

这个事务。

a. 记录binlog的时候，先记录一行 `SET @@SESSION.GTID_NEXT='server_uuid:gn0';`

b. 把这个GTID加入本实例的GTID集合。

2. 如果`gtid_next`是一个指定的GTID的值，比如通过`set gtid_next='current_gtid'`指定为`current_gtid`，那么就有两种可能：

a. 如果`current_gtid`已经存在于实例的GTID集合中，接下来执行的这个事务会直接被系统忽略；

b. 如果`current_gtid`没有存在于实例的GTID集合中，就将这个`current_gtid`分配给接下来要执行的事务，也就是说系统不需要给这个事务生成新的GTID，因此`gn0`也不用加1。

注意，一个`current_gtid`只能给一个事务使用。这个事务提交后，如果要执行下一个事务，就要执行`set`命令，把`gtid_next`设置成另外一个`gtid`或者`automatic`。

这样，每个MySQL实例都维护了一个GTID集合，用来对应“这个实例执行过的所有事务”。

这样看上去不太容易理解，接下来我就用一个简单的例子，来和你说明GTID的基本用法。

我们在实例X中创建一个表t。

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

```
insert into t values(1,1);
```

master.000001 154 Gtid	1 219 SET @@SESSION.GTID_NEXT= '00000000-1111-0000-1111-000000000000:1'
master.000001 219 Query	1 401 use `test`; CREATE TABLE `t` (
`id` int(11) NOT NULL,	
`c` int(11) DEFAULT NULL,	
PRIMARY KEY (`id`)	
) ENGINE=InnoDB	
master.000001 401 Gtid	1 466 SET @@SESSION.GTID_NEXT= '00000000-1111-0000-1111-000000000000:2'
master.000001 466 Query	1 545 BEGIN
master.000001 545 Query	1 644 use `test`; insert into t values(1,1)
master.000001 644 Xid	1 675 COMMIT /* xid=38 */

图4 初始化数据的binlog

可以看到，事务的`BEGIN`之前有一条`SET @@SESSION.GTID_NEXT`命令。这时，如果实例X有从库，那么将`CREATE TABLE`和`insert`语句的binlog同步过去执行的话，执行事务之前就会先执行这两个`SET`命令，这样被加入从库的GTID集合的，就是图中的这两个GTID。

假设，现在这个实例X是另外一个实例Y的从库，并且此时在实例Y上执行了下面这条插入语句：

```
insert into t values(1,1);
```

并且，这条语句在实例Y上的GTID是“aaaaaaaa-cccc-dddd-eeee-ffffffffffff:10”。

那么，实例X作为Y的从库，就要同步这个事务过来执行，显然会出现主键冲突，导致实例X的同步线程停止。这时，我们应该怎么处理呢？

处理方法就是，你可以执行下面的这个语句序列：

```
set gtid_next='aaaaaaaa-cccc-dddd-eeee-ffffffffffff:10';
begin;
commit;
set gtid_next=automatic;
start slave;
```

其中，前三条语句的作用，是通过提交一个空事务，把这个GTID加到实例X的GTID集合中。如图5所示，就是执行完这个空事务之后的show master status的结果。

```
mysql> show master status\G
***** 1. row *****
      File: master.000001
    Position: 885
Binlog_Do_DB:
Binlog_Ignore_DB:
Executed_Gtid_Set: 00000000-1111-0000-1111-000000000000:1-2,
aaaaaaaa-cccc-dddd-eeee-ffffffffffff:10
1 row in set (0.00 sec)
```

图5 show master status结果

可以看到实例X的Executed_Gtid_Set里面，已经加入了这个GTID。

这样，我再执行start slave命令让同步线程执行起来的时候，虽然实例X上还是会继续执行实例Y传过来的事务，但是由于“aaaaaaaa-cccc-dddd-eeee-ffffffffffff:10”已经存在于实例X的GTID集合中了，所以实例X就会直接跳过这个事务，也就不会再出现主键冲突的错误。

在上面的这个语句序列中，start slave命令之前还有一句set gtid_next=automatic。这句话的作用是“恢复GTID的默认分配行为”，也就是说如果之后有新的事务再执行，就还是按照原来的分配方式，继续分配gno=3。

基于GTID的主备切换

现在，我们已经理解GTID的概念，再一起来看看基于GTID的主备复制的用法。

在GTID模式下，备库B要设置为新主库A'的从库的语法如下：

```
CHANGE MASTER TO
MASTER_HOST=$host_name
MASTER_PORT=$port
MASTER_USER=$user_name
MASTER_PASSWORD=$password
master_auto_position=1
```

其中，`master_auto_position=1`就表示这个主备关系使用的是GTID协议。可以看到，前面让我们头疼不已的`MASTER_LOG_FILE`和`MASTER_LOG_POS`参数，已经不需要指定了。

我们把这个时刻，实例A'的GTID集合记为`set_a`，实例B的GTID集合记为`set_b`。接下来，我们就看看现在的主备切换逻辑。

我们在实例B上执行`start slave`命令，取binlog的逻辑是这样的：

1. 实例B指定主库A'，基于主备协议建立连接。
2. 实例B把`set_b`发给主库A'。
3. 实例A'算出`set_a`与`set_b`的差集，也就是所有存在于`set_a`，但是不存在于`set_b`的GTID的集合，判断A'本地是否包含了这个差集需要的所有binlog事务。
 - a.如果不包含，表示A'已经把实例B需要的binlog给删掉了，直接返回错误；
 - b.如果确认全部包含，A'从自己的binlog文件里面，找出第一个不在`set_b`的事务，发给B；
4. 之后就从这个事务开始，往后读文件，按顺序取binlog发给B去执行。

其实，这个逻辑里面包含了一个设计思想：在基于GTID的主备关系里，系统认为只要建立主备关系，就必须保证主库发给备库的日志是完整的。因此，如果实例B需要的日志已经不存在，A'就拒绝把日志发给B。

这跟基于位点的主备协议不同。基于位点的协议，是由备库决定的，备库指定哪个位点，主库就发哪个位点，不做日志的完整性判断。

基于上面的介绍，我们再来看看引入GTID后，一主多从的切换场景下，主备切换是如何实现的。

由于不需要找位点了，所以从库B、C、D只需要分别执行change master命令指向实例A'即可。

其实，严谨地说，主备切换不是不需要找位点了，而是找位点这个工作，在实例A'内部就已经自动完成了。但由于这个工作是自动的，所以对HA系统的开发人员来说，非常友好。

之后这个系统就由新主库A'写入，主库A'的自己生成的binlog中的GTID集合格式是：
server_uuid_of_A':1-M。

如果之前从库B的GTID集合格式是server_uuid_of_A:1-N，那么切换之后GTID集合的格式就变成了server_uuid_of_A:1-N, server_uuid_of_A':1-M。

当然，主库A'之前也是A的备库，因此主库A'和从库B的GTID集合是一样的。这就达到了我们预期。

GTID和在线DDL

接下来，我再举个例子帮你理解GTID。

之前在第22篇文章[《MySQL有哪些“饮鸩止渴”提高性能的方法？》](#)中，我和你提到业务高峰期的慢查询性能问题时，分析到如果是由于索引缺失引起的性能问题，我们可以通过在线加索引来解决。但是，考虑到要避免新增索引对主库性能造成的影响，我们可以先在备库加索引，然后再切换。

当时我说，在双M结构下，备库执行的DDL语句也会传给主库，为了避免传回后对主库造成影响，要通过set sql_log_bin=off关掉binlog。

评论区有位同学提出了一个问题：这样操作的话，数据库里面是加了索引，但是binlog并没有记录下这一个更新，是不是会导致数据和日志不一致？

这个问题提得非常好。当时，我在留言的回复中就引用了GTID来说明。今天，我再和你展开说明一下。

假设，这两个互为主备关系的库还是实例X和实例Y，且当前主库是X，并且都打开了GTID模式。这时的主备切换流程可以变成下面这样：

- 在实例X上执行stop slave。
- 在实例Y上执行DDL语句。注意，这里并不需要关闭binlog。
- 执行完成后，查出这个DDL语句对应的GTID，并记为server_uuid_of_Y:gn0。
- 到实例X上执行以下语句序列：

```
set GTID_NEXT="server_uuid_of_Y:gn0";
begin;
commit;
set gtid_next=automatic;
start slave;
```

这样做的目的在于，既可以让实例Y的更新有**binlog**记录，同时也可以确保不会在实例X上执行这条更新。

- 接下来，执行完主备切换，然后照着上述流程再执行一遍即可。

小结

在今天这篇文章中，我先和你介绍了一主多从的主备切换流程。在这个过程中，从库找新主库的位点是一个痛点。由此，我们引出了MySQL 5.6版本引入的**GTID**模式，介绍了**GTID**的基本概念和用法。

可以看到，在**GTID**模式下，一主多从切换就非常方便了。

因此，如果你使用的MySQL版本支持**GTID**的话，我都建议你尽量使用**GTID**模式来做一主多从的切换。

在下一篇文中，我们还能看到**GTID**模式在读写分离场景的应用。

最后，又到了我们的思考题时间。

你在**GTID**模式下设置主从关系的时候，从库执行**start slave**命令后，主库发现需要的**binlog**已经被删除掉了，导致主备创建不成功。这种情况下，你觉得可以怎么处理呢？

你可以把你的方法写在留言区，我会在下一篇文的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期间题时间

上一篇文章最后，我给你留的问题是，如果主库都是单线程压力模式，在从库追主库的过程中，**binlog-transaction-dependency-tracking** 应该选用什么参数？

这个问题的答案是，应该将这个参数设置为**WRITESET**。

由于主库是单线程压力模式，所以每个事务的**commit_id**都不同，那么设置为**COMMIT_ORDER**模式的话，从库也只能单线程执行。

同样地，由于**WRITESET_SESSION**模式要求在备库应用日志的时候，同一个线程的日志必须

与主库上执行的先后顺序相同，也会导致主库单线程压力模式下退化成单线程复制。

所以，应该将`binlog-transaction-dependency-tracking` 设置为`WRITESSET`。

评论区留言点赞板：

@慧鑫coming 问了一个好问题，对同一行作更新的几个事务，如果`commit_id`相同，是不是在备库并行执行的时候会导致数据不一致？这个问题的答案是更新同一行的事务是不可能同时进入`commit`状态的。

@老杨同志 对这个问题给出了更详细的回答，大家可以去看一下。

The image shows a promotional banner for a MySQL course. On the left, there's a logo for '极客时间' (Geek Time) with a orange play button icon. The main title 'MySQL 实战 45 讲' is displayed prominently in large, dark font. Below it, a subtitle reads '从原理到实战，丁奇带你搞懂 MySQL' (From principle to practice, Ding Qi will guide you to understand MySQL). To the right of the text, there's a portrait photo of a man with glasses and short hair, wearing a black shirt, with his arms crossed. At the bottom left, the teacher's name '林晓斌' (Lin Xiaobin) is listed along with his nickname '丁奇' and his title '前阿里资深技术专家' (Former senior technical expert at Alibaba). At the bottom, there's a call-to-action: '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。' (New version upgrade: Click 'Ask friends to read', get 10 free reads, and there are cash rewards for subscriptions).

精选留言



Mr.Strive.Z.H.L

1

老师您好：

在实际工作中，主从备份似乎是mysql用的最多的高可用方案。

但是个人认为主从备份这个方案的问题实在太多了：

1. binlog数据传输前，主库宕机，导致提交了的事务数据丢失。
2. 一主多从，即使采用半同步，也只能保证binlog至少在两台机器上，没有一个机制能够选出拥有最完整binlog的从库作为新的主库。
3. 主从切换涉及到人为操作，而不是全自动化的。即使在使用GTID的情况下，也会有binlog被删除，需要重新做从库的情况。
4. 互为主备，如果互为主备的两个实例全部宕机，mysql直接不可用。

mysql应该有更强大更完备的高可用方案（类似于zab协议或者raft协议这种），而在实际环境下，为什么主从备份用得最多呢？

2019-01-18

作者回复

3 这个应该是可以做到自动化的。

4 这个概率比较小，其实即使是别的三节点的方案，也架不住挂两个实例，所以这个不是MySQL主备的锅。

前面两点提得很对哈。

其实MySQL到现在，还是提供了很多方案可选的。很多是业务权衡的结果。

比如说，异步复制，在主库异常掉电的时候可能会丢数据。

这个大家知道以后，有一些就改成semi-sync了，但是还是有一些就留着异步复制的模式，因为semi-sync有性能影响（一开始35%，现在好点15%左右，看具体环境），而可能这些业务认为丢一两行，可以从应用层日志去补。就保留了异步复制模式。

最后，为什么主从备份用得最多，我觉得有历史原因。多年前MySQL刚要开始火的时候，大家发现这个主备模式好方便，就都用了。

而基于其他协议的方案，都是后来出现的，并且还是陆陆续续出点bug。

涉及到线上服务，大家使用新方案的热情总是局限在测试环境的多。

semi-sync也是近几年才开始稳定并被一些公司开始作为默认配置。

新技术的推广，在数据库上，确实比其他领域更需要谨慎些，也算是业务决定的吧^_^
好问题[]

以上仅一家之言哈[]

2019-01-18



某、人

1

1.如果业务允许主从不一致的情况那么可以在主上先show global variables like 'gtid_purged';然后在从上执行set global gtid_purged = '.指定从库从哪个gtid开始同步,binlog缺失那一部分,数据在从库上会丢失,就会造成主从不一致

2.需要主从数据一致的话,最好还是通过重新搭建从库来做。

3.如果有其它的从库保留有全量的binlog的话, 可以把从库指定为保留了全量binlog的从库为主库(级联复制)

4.如果binlog有备份的情况,可以先在从库上应用缺失的binlog,然后在start slave

2019-01-15

作者回复

非常好！

2019-01-15



悟空

0

看过上篇后想到一个问题：

级联复制A->B->C结构下，从库C的Seconds_Behind_Master的时间计算问题。

假定当前主库A仅有一个DDL要进行变更，耗时1分钟。那么从库C的SBM值最大应该是多少时间？是1分钟，2分钟，还是3分钟呢？

带着疑问看了一下测试从库C的binlog文件中的时间戳，得出结论应该是3分钟。

打破之前认知！请老师解惑，谢谢！

2019-01-14

| 作者回复

是的，因为算的是：当前执行时间，跟*日志时间*的差距

而这个日志时间，是在A上执行出来的。

好问题，很好的验证过程。

2019-01-14



张永志

2

今天问题回答：

GTID主从同步设置时，主库A发现需同步的GTID日志有删掉的，那么A就会报错。

解决办法：

从库B在启动同步前需要设置 gtid_purged，指定GTID同步的起点，使用备份搭建从库时需要这样设置。

如果在从库上执行了单独的操作，导致主库上缺少GTID，那么可以在主库上模拟一个与从库B上GTID一样的空事务，这样主从同步就不会报错了。

2019-01-14

| 作者回复

你已经理解GTID的机制啦！

2019-01-15



时隐时现

0

其实基于gtid复制有个大坑，在主库上千万不要执行reset master，否则从库不会报错，只会跳过gno < current_no的事务，造成一个现象就是主库复制没有中断，但是主库上的数据无法同步到从库。

2019-01-31

| 作者回复

是的，

不过reset master这种语句。。就算是基于position的协议，谁在线上主库上执行，也是直接当做删数据论处的了！

2019-01-31



Leon

从的执行是

```
CHANGE MASTER TO  
MASTER_HOST="172.27.27.2",  
MASTER_PORT=3306,  
MASTER_USER="ming",  
MASTER_PASSWORD="123456",  
master_auto_position=1;  
start slave
```

2019-01-24

0

0



Leon

老师，我这边docker起了两个mysql，一主一从
主：

```
create user 'ming'@'172.27.27.2' identified by '123456';  
GRANT REPLICATION SLAVE,RELOAD,SUPER ON *.* TO 'ming'@'%' WITH GRANT OPTION;
```

master 172.27.27.2 slave 172.27.27.3

从那边无法同步

Last_SQL_Erno: 1410

Last_SQL_Error: Error 'You are not allowed to create a user with GRANT' on query. Default database: 'test'. Query: 'GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'slave'@'%''

网上查询是授权问题，但是从容器内可以用ming的用户名和密码登录主mysql

我增加了授权还是这样，请问是什么情况

2019-01-24

0

| 作者回复

你把这个create语句直接到备库执行能执行吗？

2019-01-28



Mr.Strive.Z.H.L

0

老师您好：

之前讲过互为主备的场景下，会出现循环复制的问题，今天这节讲了GTID。

如果使用GTID，那么循环复制的问题自然而然就解决了呀？！

2019-01-18

| 作者回复

哈哈，you got it

2019-01-18



春困秋乏夏打盹

0

回答undefined的第二个问题

A-A'-B这样的级联结构

A (binlog: A:1-M)

A'(binlog: A:1-M,B:1-N) ,A'上面的操作记为B:1-N

B (binlog: A:1-M,B:1-N,C:1-X) B上面的操作记为C:1-X

---A,B,C分别为A-A'-B的uuid

2019-01-16

| 作者回复

对的

总之就是，一个主备关系里，备库的**GTID**集合应该包含主库的**GTID**集合。

2019-01-16



tchz

0

1.purge gtid, 2.重做备库数据

2019-01-15

| 作者回复

2 是ok的

purge gtid是啥

2019-01-15



fuyu

0

seta 和 setb 里的集合大小不会很大？

2019-01-15

| 作者回复

大没关系呀，是分段的，比如 `server_uuid_of_a:1-1000000`, 就一个段

2019-01-15



Leo

0

老师你好，PingCAP的大牛说分布式数据库的一个难点是时间同步。此话怎讲？mysql主从架构下时间不同步会有哪些问题？

2019-01-15

| 作者回复

今晚发布的第28篇会提到哈

2019-01-15



_CountingStars

0

老师我有一个问题 如果数据库已经有完成了很多事务 实例 A'的 GTID集合和 实例 B 的 GTID集合 是不是很大，这个GTID是从binlog里一点一点的解析出来所有的事务的吗？这样是不是会很慢？在所有binlog里定位某个GTID是不是效率也很低

2019-01-15

| 作者回复

好问题，

在binlog文件开头，有一个`Previous_gtids`, 用于记录“生成这个binlog的时候，实例的`Executed_gtid_set`”，所以启动的时候只需要解析最后一个文件；

同样的，由于有这个**Previous_gtids**，可以快速地定位GTID在哪个文件里。

2019-01-15



小超

0

老师，问个上一篇的问题，从库不是只根据binlog来做相应的操作么，这个并行复制策略根据事务相同**commit_id**判断好理解，但是根据同时进入redo log prepare 和 commit 来判断这个怎么理解？事务提交的时候，其他事务的redo log处于prepare的状态事务的某个标识也会记录到每一个事务的binlog中么？

2019-01-14



PengfeiWang

0

老师，您好： 文中对于**sql_slave_skip_counter=1**的理解似乎有偏差，官方文档中的解释是：
When you use SET GLOBAL sql_slave_skip_counter to skip events and the result is in the middle of a group, the slave continues to skip events until it reaches the end of the group. Execution then starts with the next event group.

按照官方文档的解释，命令**sql_slave_skip_counter=1** 应该是跳过一个事务中的1个event，除非这个事务是有单个event组成的，才会跳过一个事务。

2019-01-14

| 作者回复

你这个是好问题，

确实只是跳过一个event，不过文档中说了呀

“the slave continues to skip events until it reaches the end of the group.”，

所以效果上等效于跳过一个事务哦

2019-01-14



PengfeiWang

0

老师，你好： 在生产环境（基于位点的主备切换）中，经常会遇到这样的场景：备库由于硬件或其他原因异常宕机，恢复后重启备库，执行**start slave**命令，总会遇到**1062**主键重复的报错，一直解释不清楚为什么？

2019-01-14

| 作者回复

看一下这个语句的结果，会受这几个参数的影响哈

```
select * from information_schema.GLOBAL_VARIABLES where VARIABLE_NAME in ('master_info_repository','relay_log_info_repository','sync_master_info','sync_relay_log_info', 'sync_binlog', 'innodb_flush_log_at_trx_commit');
```

2019-01-14



路过

0

老师，请教：

show slave status\G的输出中，包含如下：

Executed_Gtid_Set: 572ece6c-e3ed-11e8-92c4-005056a509d8:1-1136659,

ecb34895-e3eb-11e8-80e9-005056a55d62:1-1015

是不是表示当前slave曾经和两个master同步过？

2019-01-14

| 作者回复

一个是它自己吧？

select @@server_uuid 看看

2019-01-14



undefined

0

老师有几个问题：

1. 会不会出现主库切换后，B 中已经执行过的事务，而 A' 由于网络延迟还没有收到，此时已经对 B 执行切换主库，这时候，B 中有该 GTID，但是 A' 中没有，这种情况会怎么处理
2. 如果 A 是主库，A' 备库，B 是 A' 的从库，此时 B 的 GTID 集合应该是 server_uuid_of_A:1-N，此时 A' 宕机，B 改为监听 A，这时候 A 和 B 的 GTID 集合没有交集，会不会发生 A 将所有的 binlog 重新发给 B
3. 思考题我的理解是从主库中 dump 出相关的数据，在备库中执行后再次执行 start slave；评论中说到从其他从库获取，但是如果只有一主一从，有 binlog 丢失，是不是只要 dump 文件恢复这一个办法

2019-01-14

| 作者回复

1. 这个也是异步复制导致的，只有 semi-sync 能解了。。

2. 不是哦，如果“A 是主库，A' 备库，B 是 A' 的从库”，那所有 A 的更新也都会通过 A 传给 B，所以 B 的 GTID 集合正常就是包含了 A 和 A' 的

3. “如果只有一主一从，有 binlog 丢失”，是的，就只有备库重做了

2019-01-16



亮

0

老师您好，假如 a 宕机了，需要把从切换到 a'，这时候业务已经有感知了吧？怎么能让业务尽量没有感知呢？谢谢老师

2019-01-14

| 作者回复

这种情况下，不可能业务完全无感知，

但是如果业务代码有“重连并重试”的逻辑，并且切换足够快，就可以对业务无影响，前提是解决主备延迟问题，就是 25、26 两篇提到的

2019-01-14



大坤

0

今天问题回答，由于 GTID 具有全局唯一性，那么其它正常的 gtid 已经被复制到了其他从库上了，只需要切换 gtid 到其他从库，等待同步完毕后在切换回主库即可

2019-01-14

| 作者回复

这个想法很不错』

2019-01-14

28 | 读写分离有哪些坑？

2019-01-16 林晓斌



在上一篇文章中，我和你介绍了一主多从的结构以及切换流程。今天我们就继续聊聊一主多从架构的应用场景：读写分离，以及怎么处理主备延迟导致的读写分离问题。

我们在上一篇文章中提到的一主多从的结构，其实就是读写分离的基本结构了。这里，我再把这张图贴过来，方便你理解。

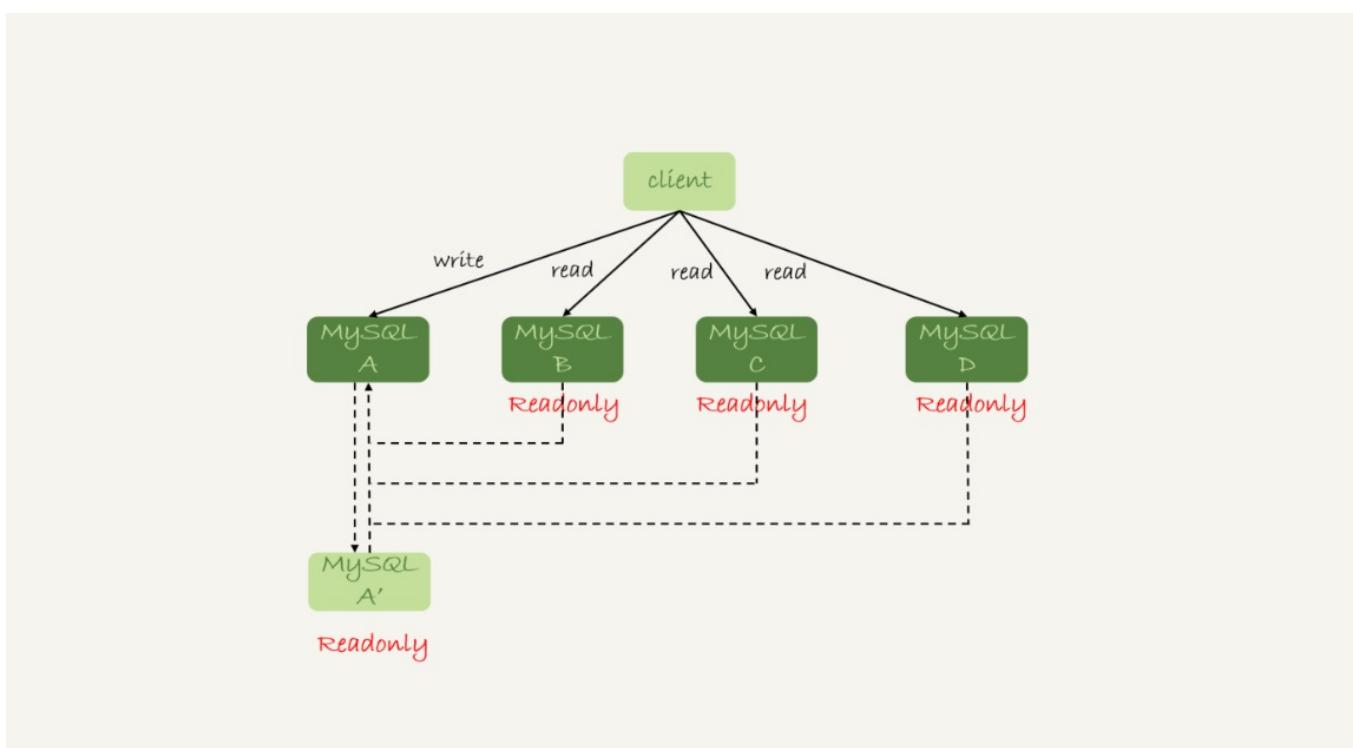


图1 读写分离基本结构

读写分离的主要目标就是分摊主库的压力。图1中的结构是客户端（client）主动做负载均衡，这种模式下一般会把数据库的连接信息放在客户端的连接层。也就是说，由客户端来选择后端数据库进行查询。

还有一种架构是，在MySQL和客户端之间有一个中间代理层proxy，客户端只连接proxy，由proxy根据请求类型和上下文决定请求的分发路由。

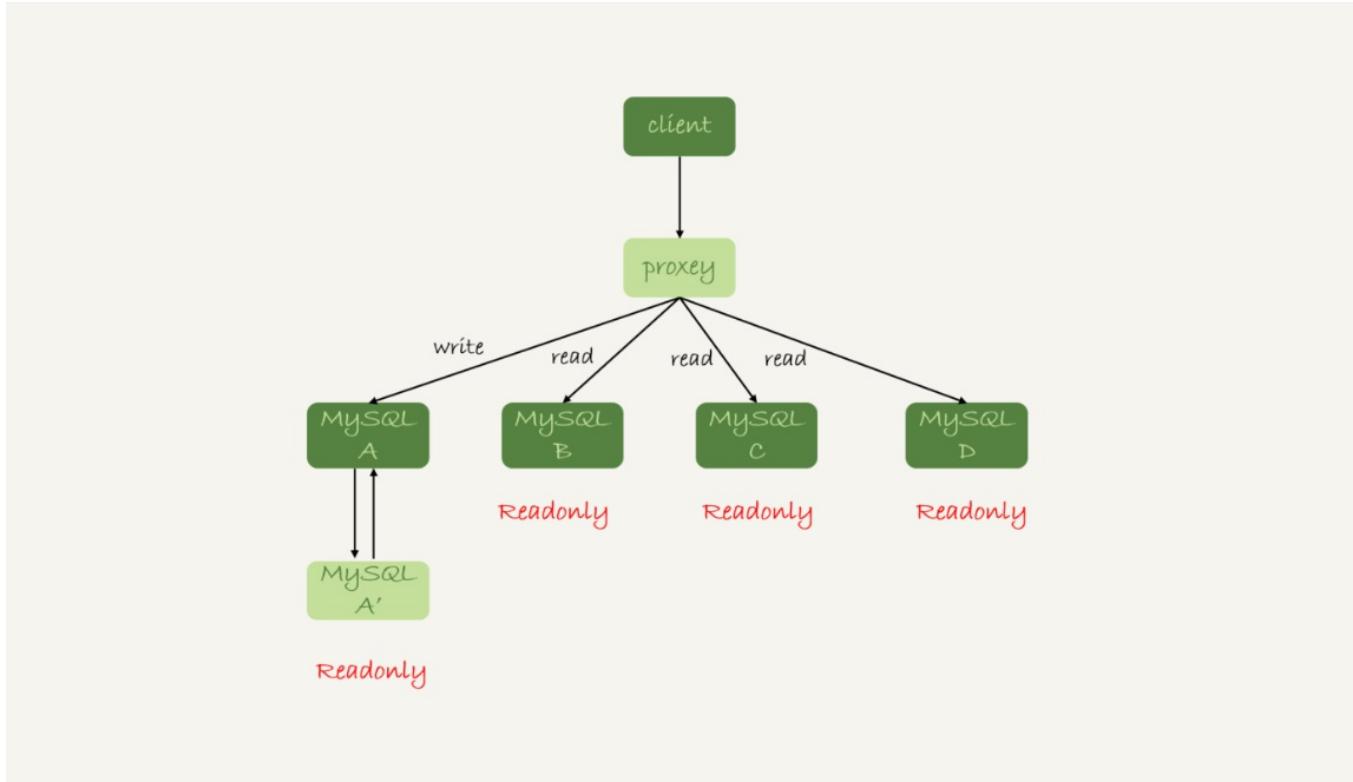


图2 带proxy的读写分离架构

接下来，我们就看一下客户端直连和带proxy的读写分离架构，各有哪些特点。

1. 客户端直连方案，因为少了一层proxy转发，所以查询性能稍微好一点儿，并且整体架构简单，排查问题更方便。但是这种方案，由于要了解后端部署细节，所以在出现主备切换、库迁移等操作的时候，客户端都会感知到，并且需要调整数据库连接信息。
你可能会觉得这样客户端也太麻烦了，信息大量冗余，架构很丑。其实也未必，一般采用这样的架构，一定会伴随一个负责管理后端的组件，比如Zookeeper，尽量让业务端只专注于业务逻辑开发。
2. 带proxy的架构，对客户端比较友好。客户端不需要关注后端细节，连接维护、后端信息维护等工作，都是由proxy完成的。但这样的话，对后端维护团队的要求会更高。而且，proxy也需要有高可用架构。因此，带proxy架构的整体就相对比较复杂。

理解了这两种方案的优劣，具体选择哪个方案就取决于数据库团队提供的能力了。但目前看，趋势是往带proxy的架构方向发展的。

但是，不论使用哪种架构，你都会碰到我们今天要讨论的问题：由于主从可能存在延迟，客户端执行完一个更新事务后马上发起查询，如果查询选择的是从库的话，就有可能读到刚刚的事务更新之前的状态。

这种“在从库上会读到系统的一个过期状态”的现象，在这篇文章里，我们暂且称之为“过期读”。

前面我们说过了几种可能导致主备延迟的原因，以及对应的优化策略，但是主从延迟还是不能100%避免的。

不论哪种结构，客户端都希望查询从库的数据结果，跟查主库的数据结果是一样的。

接下来，我们就来讨论怎么处理过期读问题。

这里，我先把文章中涉及到的处理过期读的方案汇总在这里，以帮助你更好地理解和掌握全文的知识脉络。这些方案包括：

- 强制走主库方案；
- sleep方案；
- 判断主备无延迟方案；
- 配合semi-sync方案；
- 等主库位点方案；
- 等GTID方案。

强制走主库方案

强制走主库方案其实就是，将查询请求做分类。通常情况下，我们可以将查询请求分为这么两类：

1. 对于必须要拿到最新结果的请求，强制将其发到主库上。比如，在一个交易平台上，卖家发布商品以后，马上要返回主页面，看商品是否发布成功。那么，这个请求需要拿到最新的结果，就必须走主库。
2. 对于可以读到旧数据的请求，才将其发到从库上。在这个交易平台上，买家来逛商铺页面，就算晚几秒看到最新发布的商品，也是可以接受的。那么，这类请求就可以走从库。

你可能会说，这个方案是不是有点畏难和取巧的意思，但其实这个方案是用得最多的。

当然，这个方案最大的问题在于，有时候你会碰到“所有查询都不能是过期读”的需求，比如一些金融类的业务。这样的话，你就要放弃读写分离，所有读写压力都在主库，等同于放弃了扩展性。

因此接下来，我们来讨论的话题是：可以支持读写分离的场景下，有哪些解决过期读的方案，并分析各个方案的优缺点。

Sleep 方案

主库更新后，读从库之前先sleep一下。具体的方案就是，类似于执行一条select sleep(1)命令。

这个方案的假设是，大多数情况下主备延迟在1秒之内，做一个sleep可以有很大概率拿到最新的数据。

这个方案给你的第一感觉，很可能是不靠谱儿，应该不会有人用吧？并且，你还可能会说，直接在发起查询时先执行一条sleep语句，用户体验很不友好啊。

但，这个思路确实可以在一定程度上解决问题。为了看起来更靠谱儿，我们可以换一种方式。

以卖家发布商品为例，商品发布后，用Ajax（Asynchronous JavaScript + XML，异步JavaScript和XML）直接把客户端输入的内容作为“新的商品”显示在页面上，而不是真正地去数据库做查询。

这样，卖家就可以通过这个显示，来确认产品已经发布成功了。等到卖家再刷新页面，去查看商品的时候，其实已经过了一段时间，也就达到了sleep的目的，进而也就解决了过期读的问题。

也就是说，这个sleep方案确实解决了类似场景下的过期读问题。但，从严格意义上来说，这个方案存在的问题就是不精确。这个不精确包含了两层意思：

1. 如果这个查询请求本来0.5秒就可以在从库上拿到正确结果，也会等1秒；
2. 如果延迟超过1秒，还是会出现过期读。

看到这里，你是不是有一种“你是不是在逗我”的感觉，这个改进方案虽然可以解决类似Ajax场景下的过期读问题，但还是怎么看都不靠谱儿。别着急，接下来我就和你介绍一些更准确的方案。

判断主备无延迟方案

要确保备库无延迟，通常有三种做法。

通过前面的[第25篇文章](#)，我们知道show slave status结果里的seconds_behind_master参数的值，可以用来衡量主备延迟时间的长短。

所以第一种确保主备无延迟的方法是，每次从库执行查询请求前，先判断seconds_behind_master是否已经等于0。如果还不等于0，那就必须等到这个参数变为0才能执行查询请求。

seconds_behind_master的单位是秒，如果你觉得精度不够的话，还可以采用对比位点和GTID

的方法来确保主备无延迟，也就是我们接下来要说的第二和第三种方法。

如图3所示，是一个`show slave status`结果的部分截图。

```
Master_Log_File: master.000012
Read_Master_Log_Pos: 126067593
.....
Relay_Master_Log_File: master.000012
.....
Exec_Master_Log_Pos: 126067593
.....
Retrieved_Gtid_Set: 00000000-1111-0000-1111-000000000000:1-10000
Executed_Gtid_Set: 00000000-1111-0000-1111-000000000000:1-10000
Auto_Position: 1
```

图3 show slave status结果

现在，我们就通过这个结果，来看看具体如何通过对比位点和GTID来确保主备无延迟。

第二种方法，对比位点确保主备无延迟：

- `Master_Log_File`和`Read_Master_Log_Pos`, 表示的是读到的主库的最新位点;
- `Relay_Master_Log_File`和`Exec_Master_Log_Pos`, 表示的是备库执行的最新位点。

如果`Master_Log_File`和`Relay_Master_Log_File`、`Read_Master_Log_Pos`和`Exec_Master_Log_Pos`这两组值完全相同，就表示接收到的日志已经同步完成。

第三种方法，对比GTID集合确保主备无延迟：

- `Auto_Position=1`，表示这对主备关系使用了GTID协议。
- `Retrieved_Gtid_Set`, 是备库收到的所有日志的GTID集合;
- `Executed_Gtid_Set`, 是备库所有已经执行完成的GTID集合。

如果这两个集合相同，也表示备库接收到的日志都已经同步完成。

可见，对比位点和对比GTID这两种方法，都要比判断`seconds_behind_master`是否为0更准确。

在执行查询请求之前，先判断从库是否同步完成的方法，相比于`sleep`方案，准确度确实提升了不少，但还是没有达到“精确”的程度。为什么这么说呢？

我们现在一起来回顾下，一个事务的binlog在主备库之间的状态：

1. 主库执行完成，写入binlog，并反馈给客户端；
2. binlog被从主库发送给备库，备库收到；
3. 在备库执行binlog完成。

我们上面判断主备无延迟的逻辑，是“备库收到的日志都执行完成了”。但是，从binlog在主备之间状态的分析中，不难看出还有一部分日志，处于客户端已经收到提交确认，而备库还没收到日志的状态。

如图4所示就是这样的一个状态。

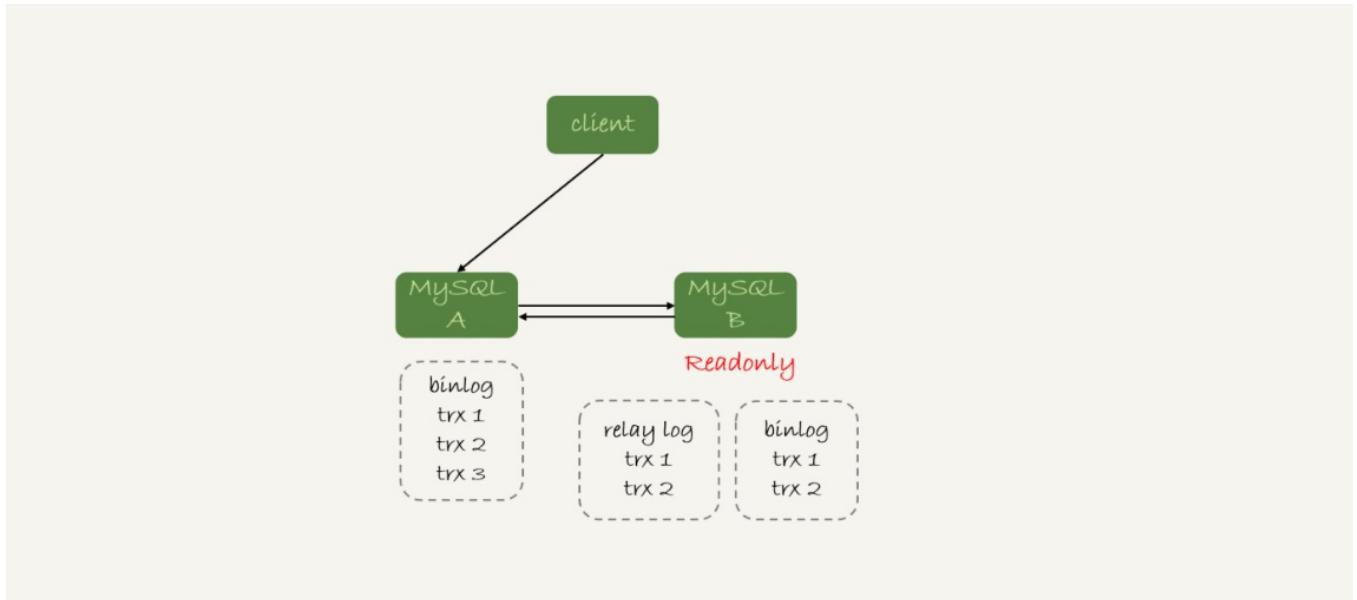


图4 备库还没收到trx3

这时，主库上执行完成了三个事务trx1、trx2和trx3，其中：

1. trx1和trx2已经传到从库，并且已经执行完成了；
2. trx3在主库执行完成，并且已经回复给客户端，但是还没有传到从库中。

如果这时候你在从库B上执行查询请求，按照我们上面的逻辑，从库认为已经没有同步延迟，但还是查不到trx3的。严格地说，就是出现了过期读。

那么，这个问题有没有办法解决呢？

配合semi-sync

要解决这个问题，就要引入半同步复制，也就是semi-sync replication。

semi-sync做了这样的设计：

1. 事务提交的时候，主库把binlog发给从库；
2. 从库收到binlog以后，发回给主库一个ack，表示收到了；
3. 主库收到这个ack以后，才能给客户端返回“事务完成”的确认。

也就是说，如果启用了semi-sync，就表示所有给客户端发送过确认的事务，都确保了备库已经

收到了这个日志。

在[第25篇文章](#)的评论区，有同学问到：如果主库掉电的时候，有些binlog还来不及发给从库，会不会导致系统数据丢失？

答案是，如果使用的是普通的异步复制模式，就可能会丢失，但**semi-sync**就可以解决这个问题。

这样，**semi-sync**配合前面关于位点的判断，就能够确定在从库上执行的查询请求，可以避免过期读。

但是，**semi-sync+位点判断**的方案，只对一主一备的场景是成立的。在一主多从场景中，主库只要等到一个从库的**ack**，就开始给客户端返回确认。这时，在从库上执行查询请求，就有两种情况：

1. 如果查询是落在这个响应了**ack**的从库上，是能够确保读到最新数据；
2. 但如果是查询落到其他从库上，它们可能还没有收到最新的日志，就会产生过期读的问题。

其实，判断同步位点的方案还有另外一个潜在的问题，即：如果在业务更新的高峰期，主库的位点或者**GTID**集合更新很快，那么上面的两个位点等值判断就会一直不成立，很可能出现从库上迟迟无法响应查询请求的情况。

实际上，回到我们最初的业务逻辑里，当发起一个查询请求以后，我们要得到准确的结果，其实并不需要等到“主备完全同步”。

为什么这么说呢？我们来看一下这个时序图。

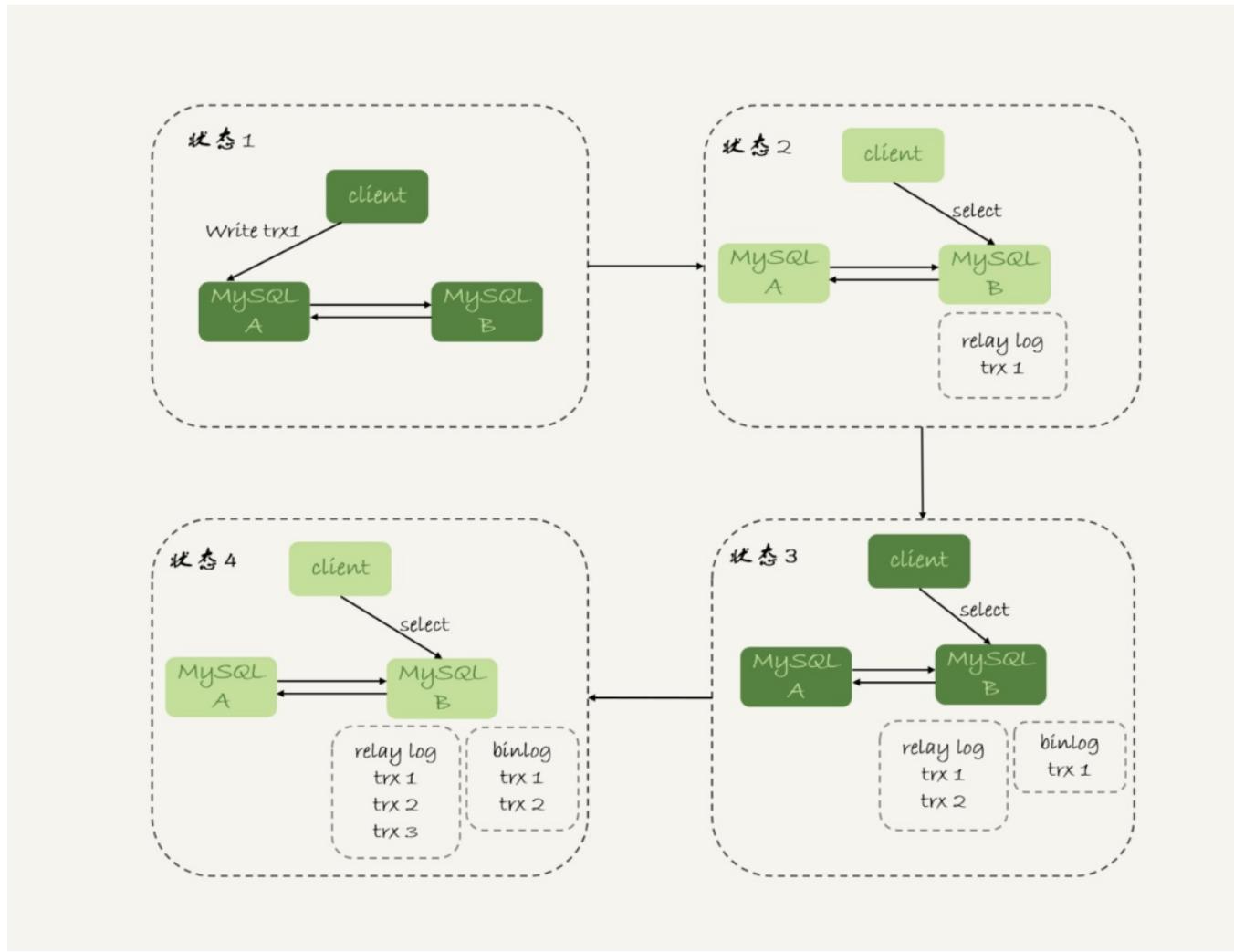


图5 主备持续延迟一个事务

图5所示，就是等待位点方案的一个bad case。图中备库B下的虚线框，分别表示relaylog和binlog中的事务。可以看到，图5中从状态1到状态4，一直处于延迟一个事务的状态。

备库B一直到状态4都和主库A存在延迟，如果用上面必须等到无延迟才能查询的方案，`select`语句直到状态4都不能被执行。

但是，其实客户端是在发完`trx1`更新后发起的`select`语句，我们只需要确保`trx1`已经执行完成就可以执行`select`语句了。也就是说，如果在状态3执行查询请求，得到的就是预期结果了。

到这里，我们小结一下，`semi-sync`配合判断主备无延迟的方案，存在两个问题：

1. 一主多从的时候，在某些从库执行查询请求会存在过期读的现象；
2. 在持续延迟的情况下，可能出现过度等待的问题。

接下来，我要和你介绍的等主库位点方案，就可以解决这两个问题。

等主库位点方案

要理解等主库位点方案，我需要先和你介绍一条命令：

```
select master_pos_wait(file, pos[, timeout]);
```

这条命令的逻辑如下：

1. 它是在从库执行的；
2. 参数**file**和**pos**指的是主库上的文件名和位置；
3. **timeout**可选，设置为正整数**N**表示这个函数最多等待**N**秒。

这个命令正常返回的结果是一个正整数**M**，表示从命令开始执行，到应用完**file**和**pos**表示的**binlog**位置，执行了多少事务。

当然，除了正常返回一个正整数**M**外，这条命令还会返回一些其他结果，包括：

1. 如果执行期间，备库同步线程发生异常，则返回**NULL**；
2. 如果等待超过**N**秒，就返回**-1**；
3. 如果刚开始执行的时候，就发现已经执行过这个位置了，则返回**0**。

对于图5中先执行**trx1**，再执行一个查询请求的逻辑，要保证能够查到正确的数据，我们可以使用这个逻辑：

1. **trx1**事务更新完成后，马上执行**show master status**得到当前主库执行到的**File**和**Position**；
2. 选定一个从库执行查询语句；
3. 在从库上执行**select master_pos_wait(File, Position, 1)**；
4. 如果返回值是**>=0**的正整数，则在这个从库执行查询语句；
5. 否则，到主库执行查询语句。

我把上面这个流程画出来。

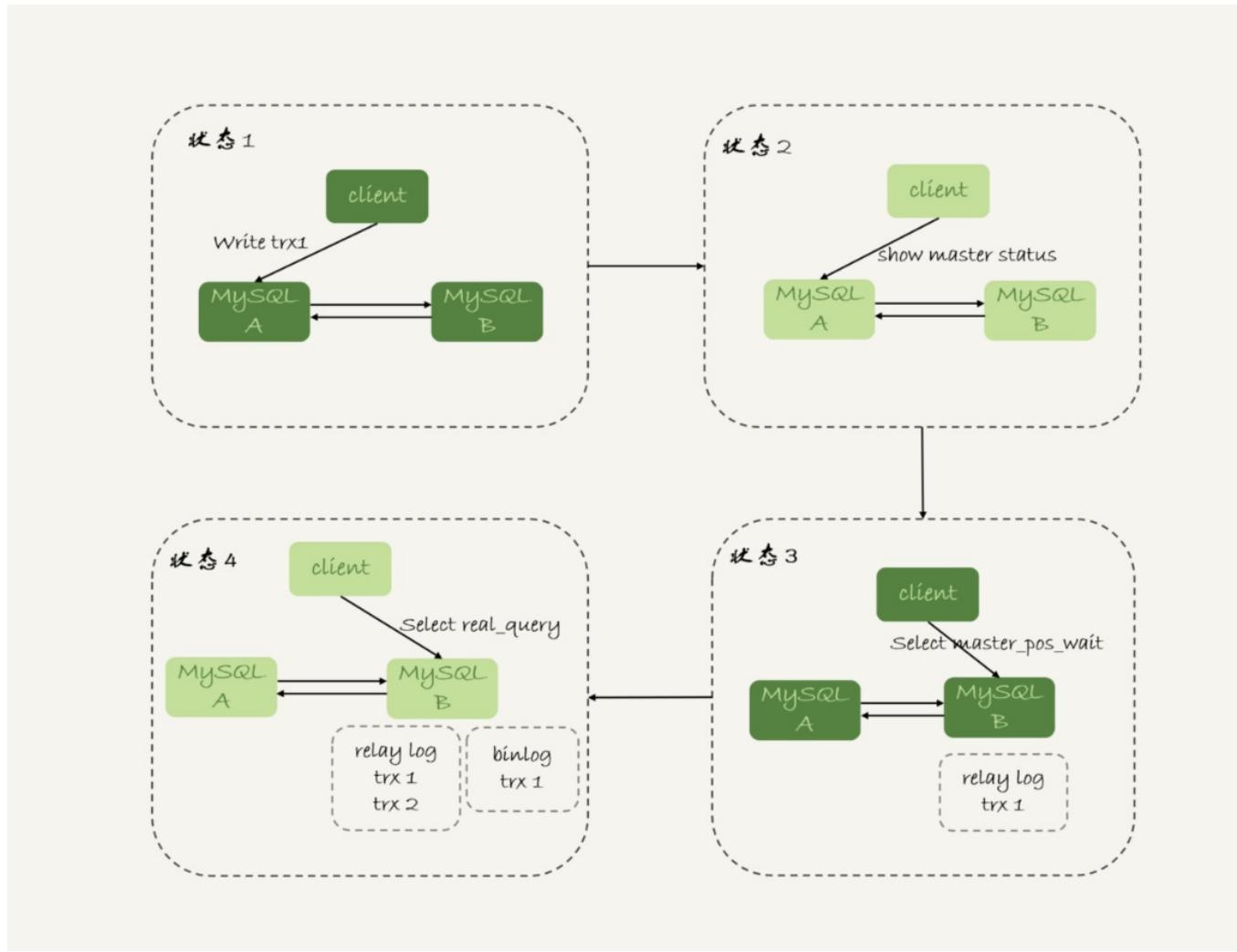


图6 master_pos_wait方案

这里我们假设，这条**select**查询最多在从库上等待1秒。那么，如果1秒内**master_pos_wait**返回一个大于等于0的整数，就确保了从库上执行的这个查询结果一定包含了**trx1**的数据。

步骤5到主库执行查询语句，是这类方案常用的退化机制。因为从库的延迟时间不可控，不能无限等待，所以如果等待超时，就应该放弃，然后到主库去查。

你可能会说，如果所有的从库都延迟超过1秒了，那查询压力不就都跑到主库上了吗？确实是这样。

但是，按照我们设定不允许过期读的要求，就只有两种选择，一种是超时放弃，一种是转到主库查询。具体怎么选择，就需要业务开发同学做好限流策略了。

GTID方案

如果你的数据库开启了**GTID**模式，对应的也有等待**GTID**的方案。

MySQL中同样提供了一个类似的命令：

```
select wait_for_executed_gtid_set(gtid_set, 1);
```

这条命令的逻辑是：

1. 等待，直到这个库执行的事务中包含传入的gtid_set，返回0；
2. 超时返回1。

在前面等位点的方案中，我们执行完事务后，还要主动去主库执行**show master status**。而MySQL 5.7.6版本开始，允许在执行完更新类事务后，把这个事务的**GTID**返回给客户端，这样等**GTID**的方案就可以减少一次查询。

这时，等**GTID**的执行流程就变成了：

1. **trx1**事务更新完成后，从返回包直接获取这个事务的**GTID**，记为**gtid1**；
2. 选定一个从库执行查询语句；
3. 在从库上执行 **select wait_for_executed_gtid_set(gtid1, 1)**；
4. 如果返回值是0，则在这个从库执行查询语句；
5. 否则，到主库执行查询语句。

跟等主库位点的方案一样，等待超时后是否直接到主库查询，需要业务开发同学来做限流考虑。

我把这个流程图画出来。

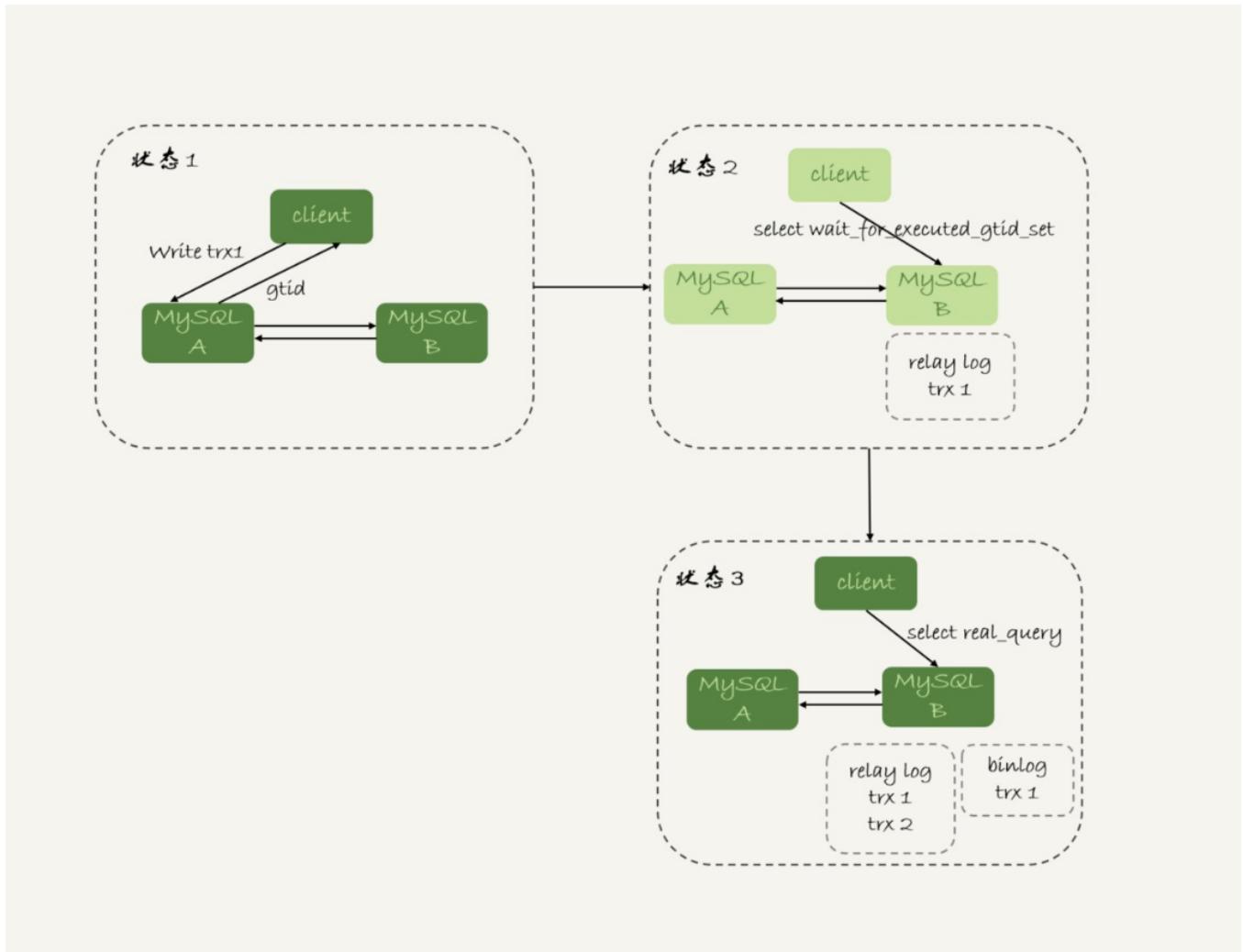


图7 `wait_for_executed_gtid_set`方案

在上面的第一步中，`trx1`事务更新完成后，从返回包直接获取这个事务的GTID。问题是，怎么能够让MySQL在执行事务后，返回包中带上GTID呢？

你只需要将参数`session_track_gtids`设置为`OWN_GTID`，然后通过API接口`mysql_session_track_get_first`从返回包解析出GTID的值即可。

在专栏的[第一篇文章](#)中，我介绍`mysql_reset_connection`的时候，评论区有同学留言问这类接口应该怎么使用。

这里我再回答一下。其实，MySQL并没有提供这类接口的SQL用法，是提供给程序的API(<https://dev.mysql.com/doc/refman/5.7/en/c-api-functions.html>)。

比如，为了让客户端在事务提交后，返回的GTID能够在客户端显示出来，我对MySQL客户端代码做了点修改，如下所示：

```
const char *data;
size_t length;
if (mysql_session_track_get_first(&mysql, SESSION_TRACK_GTIDS, &data, &length) == 0)
{
    sprintf(buff, "GTID: %s", data);
}
```

图8 显示更新事务的GTID—代码

这样，就可以看到语句执行完成，显示出GTID的值。

```
mysql> set session_track_gtids=OWN_GTID;
Query OK, 0 rows affected (0.00 sec)

mysql> update t set c=c+1;
Query OK, 0 rows affected, GTID: 00000000-1111-0000-1111-000000000000:9 (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

图9 显示更新事务的GTID—效果

当然了，这只是一个例子。你要使用这个方案的时候，还是应该在你的客户端代码中调用`mysql_session_track_get_first`这个函数。

小结

在今天这篇文章中，我跟你介绍了一主多从做读写分离时，可能碰到过期读的原因，以及几种应对的方案。

这几种方案中，有的方案看上去是做了妥协，有的方案看上去不那么靠谱儿，但都是有实际应用场景的，你需要根据业务需求选择。

即使是最后的等待位点和等待GTID这两个方案，虽然看上去比较靠谱儿，但仍然存在需要权衡的情况。如果所有的从库都延迟，那么请求就会全部落到主库上，这时候会不会由于压力突然增大，把主库打挂了呢？

其实，在实际应用中，这几个方案是可以混合使用的。

比如，先在客户端对请求做分类，区分哪些请求可以接受过期读，而哪些请求完全不能接受过期读；然后，对于不能接受过期读的语句，再使用等GTID或等位点的方案。

但话说回来，过期读在本质上是由一写多读导致的。在实际应用中，可能会有别的不需要等待就可以水平扩展的数据库方案，但这往往是用牺牲写性能换来的，也就是需要在读性能和写性能中取权衡。

最后，我给你留下一个问题吧。

假设你的系统采用了我们文中介绍的最后一个方案，也就是等GTID的方案，现在你要对主库的一张大表做DDL，可能会出现什么情况呢？为了避免这种情况，你会怎么做呢？

你可以把你的分析和方案设计写在评论区，我会在下一篇文章跟你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期给你留的问题是，在GTID模式下，如果一个新的从库接上主库，但是需要的binlog已经没了，要怎么做？

@某、人同学给了很详细的分析，我把他的回答略做修改贴过来。

1. 如果业务允许主从不一致的情况，那么可以在主库上先执行`show global variables like 'gtid_purged'`，得到主库已经删除的GTID集合，假设是`gtid_purged1`；然后先在从库上执行`reset master`，再执行`set global gtid_purged = 'gtid_purged1'`；最后执行`start slave`，就会从主库现存的binlog开始同步。binlog缺失的那一部分，数据在从库上就可能会有丢失，造成主从不一致。
2. 如果需要主从数据一致的话，最好还是通过重新搭建从库来做。
3. 如果有其他的从库保留有全量的binlog的话，可以把新的从库先接到这个保留了全量binlog的从库，追上日志以后，如果有需要，再接回主库。
4. 如果binlog有备份的情况，可以先在从库上应用缺失的binlog，然后再执行`start slave`。

评论区留言点赞板：

@悟空 同学级联实验，验证了`seconds_behind_master`的计算逻辑。

@_CountingStars 问了一个好问题：MySQL是怎么快速定位binlog里面的某一个GTID位置的？答案是，在binlog文件头部的`Previous_gtids`可以解决这个问题。

@王朋飞 同学问了一个好问题，`sql_slave_skip_counter`跳过的是一个event，由于MySQL总不能执行一半的事务，所以既然跳过了一个event，就会跳到这个事务的末尾，因此`set global sql_slave_skip_counter=1;start slave`是可以跳过整个事务的。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「」，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



有铭

6

这专栏真的是干货满满，每看一篇我都有“我发现我真的不会使用MySQL”和“我原来把MySQL用错了”的挫败感

2019-01-16

| 作者回复

这样我觉得你和我的时间都值了！

把你更新了认识的点发到评论区，这样会印象更深哈！

2019-01-16



某、人

3

老师我先请教两个问题(估计大多数同学都有这个疑惑)：

- 1.现在的中间件可以说是乱花渐欲迷人眼,请问老师哪一款中间件适合大多数不分库分表,只是做读写分离业务的proxy,能推荐一款嘛?毕竟大多数公司都没有专门做中间件开发的团队
- 2.如果是业务上进行了分库分表,老师能推荐一款分库分表的proxy嘛?我目前了解到的针对分库分表的proxy都或多或少有些问题。不过分布式数据库是一个趋势也是一个难点。

2019-01-16

| 作者回复

额，这个最难回答了

说实话因为我原来团队是团队自己做的proxy（没有开源），所以我对其他proxy用得并不多，实在不敢随便指一个。

如果说个比较熟悉的话，可能MariaDB MaxScale还不错

2019-01-17



曾剑

2

老师写的每一篇文章都能让我获益良多。每一篇都值得看好几遍。

今天的问题，大表做DDL的时候可能会出现主从延迟，导致等 GTID 的方案可能会导致这部分流量全打到主库，或者全部超时。

如果这部分流量太大的话，我会选择上一篇文章介绍的两种方法：

1.在各个从库先SET sql_log_bin = OFF，然后做DDL，所有从库及备主全做完之后，做主从切换，最后在原来的主库用同样的方式做DDL。

2.从库上执行DDL；将从库上执行DDL产生的GTID在主库上利用生成一个空事务GTID的方式将这个GTID在主库上生成出来。

各个从库做完之后再主从切换，然后再在原来的主库上同样做一次。

需要注意的是如果有MM架构的情况下，承担写职责的主库上的slave需要先停掉。

2019-01-16

| 作者回复

回复表示这两篇文章你都get到了

2019-01-16



二马

2

最近做性能测试时发现当并发用户达到一定量(比如500)，部分用户连接不上，能否介绍下MySQL连接相关问题，谢谢！

2019-01-16

| 作者回复

修改max_connections参数

2019-01-16



IceGeek17

1

老师，能不能分析下，如果去实现一个做读写分离的proxy，有哪些重要的点要考虑，比如：连接管理、流量分配管理、proxy自己的高可用，等等。

因为老师原来的团队自己开发过proxy，肯定有相关的经验，也趟过很多坑，能不能从如何实现一个proxy需要考虑哪些关键点，在架构上做一个分析和梳理

2019-01-29

| 作者回复

额，这个问题有点大... 你提一个具体问题我们来讨论吧

2019-01-31



猪哥哥

1

老师，你真棒，我公司的生产环境解决过期读使用的就是强制走主库方案，看了这篇文章，困惑了很久的问题迎刃而解！很感谢！

2019-01-17



易翔

为老师一句你的时间和我的时间都值了。点赞

1

2019-01-16



Mr.Strive.Z.H.L

0

老师您好：

关于主库大表的**DDL**操作，我看了问题答案，有两种方案。第一种是读写请求转到主库，在主库上做**DDL**。第二种是从库上做**DDL**，完成后进行主从切换。

关于第二种，有一个疑惑：

从库上做**DDL**，读写请求走主库，等到从库完成后，从库必须要同步**DDL**期间，主库完成的事务后才能进行主从切换。而如果**DDL**操作是删除一列，那么在同步过程中会出错呀？（比如抛出这一列不存在的错误）。

2019-01-21

| 作者回复

你说得对，这种方案下能支持的**DDL**只有以下几种：

创建/删除索引、新增最后一列、删除最后一列

其中**DBA**会认为“合理”的**DDL**需求就是：“创建/删除索引、新增最后一列”

新春快乐~

2019-02-04



black_mirror

0

林老师 您好

1.`mysql_session_track_get_fitst`这个函数，从github下载mysql源码后怎么尝试简单编译，如图8?

2. `mysql_session_track_get_fitst`这个函数貌似不支持python语言，我想模拟文中等gtid方法，不会java怎么办？

2019-01-21



black_mirror

0

林老师 您好

请问`mysql_session_track_get_fitst`这个函数查询了官方资料都需要可以修改源码

1.在不懂c++情况下，github上下载源码后怎么尝试简单编译使用，如图8代码

2. `mysql_session_track_get_fitst`函数貌似没有python语言api，不会java，想在代码层面模拟整个过程，还有木有解决方法？

2019-01-21

| 作者回复

不知道python是不是有方法可以把c代码作为扩展模块

2019-01-21



...

-



辣椒

0

老师，`mysql_session_track_get_first`是c的，有没有java的？

2019-01-18

| 作者回复

额，这个我还真不知道，抱歉哈。

2019-01-18



信信

0

老师您好，文中判断主备无延迟方案的第二种和第三种方法，都是对比了主从执行完的日志是否相同。因为不会出现图4下方说的：“从库认为已经没有同步延迟，但还是查不到 `trx3` 的。”因为如果从库未执行 `trx3` 的话，第二，第三种方法都是不通过的。

2019-01-18

| 作者回复

不会哦

如果 `trx3` 还没传到备库，备库是会认为已经同步完成了

2019-01-18



Max

0

我一般是先是在从库上设置 `set_log_bin=off`，然后执行 `ddl` 语句。

然后完成以后，主从做一下切换。然后在主库上在执行一下 `set_log_bin=off`，执行 `ddl` 语句。

然后在做一下主从切换。

个人对 `pt-online-schema-change` 不是很推荐使用，它的原理基本是创建触发器，然后创建和旧表一样结构的数据表，

把旧表的数据复制过去。最后删除旧表。以前做个一个测试，如果旧表一直在被 `select`，删除过程会一直会等待。

所以个人不是很建议。万一不小心变成从删库到路步，那就得不偿失了。

老师，有个问题想请教一下，一主多从可以多到什么地步，以前我们CTO解决的方案就是加机器，一主十三从。

当时我是反对的，其实个人建议还是从 `SQL`，业务上面去优化。而不是一味的加机器。如果加机器解决的话，还要 `DBA` 做什么呢？

2019-01-17

| 作者回复

前面的分析很好哈

然后一主13从有点多了，否则主库生成 `binlog` 太快的话，主库的网卡会被打爆。要这么多的话，得做级联。

`DBA` 解决不能靠加机器解决的事情^_^ 而且如果通过优化，可以把13变成3，那也是 `DBA` 的价值

2019-01-17



coderfocus

0

出来挨打

一步一步 循序渐渐 讲的太棒了 感谢老师

2019-01-17



ThinkingQuest

0

楼上有人提到8小时自动断开连接的问题。

mysql中有`wait_timeout`和`interactive_timeout`两个参数。

这俩参数挺容易混淆的，往上博客文章说的很多，但是不敢相信他们。

官方的解释在这里：

https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_interactive_timeout

https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_wait_timeout

只说用这两个参数中的哪个，取决于客户端调用`mysql_real_connect()`的时候传递的`options`中是否使用了`CLIENT_INTERACTIVE`选项。

但是很多做java开发的同学，想必并不知道JDBC的connector用的是哪一个。

我倾向于认为是`interactive_timeout`。 mysql client cli大概是`wait_timeout`吧。

其实做一个实验就可以知道结果。但是不阅读mysql代码，大概不能理解mysql为什么设计这么两个`timeout`，是出于什么考虑的。

2019-01-17

| 作者回复

JDBC的connector我也没研究过，不过我认为应该是非`interactive`模式。

需要这两个的原因，还是因为有不同的使用模式，给MySQL客户端和一些其他的可视化工具客户端使用。

2019-01-17



不停

0

高手，你这现在在哪里上班了，有兴趣来我们公司耍耍么？

2019-01-16



万勇

0

老师，请教下。

1.对大表做`ddl`，是可以采用先在备库上`set global log_bin=off`，先做完`ddl`，然后切换主备库。为了保证数据一致性，在切主备的时候，数据库会有个不可用的时间段，对业务会造成影响。现在的架构方式，中间层还有`proxy`，意味着`proxy`也需要修改主备配置，做`reload`。这样做的话，感觉成本太高，在真正的生产环境中，这种方法适用吗？

2.目前我们常采用的是对几百万以上的表用`pt-online-schema-change`，这种方式会产生大量的`b inlog`，业务高峰期不能做，会引起主备延迟。在生产业务中，我觉得等主库节点或者等`gtid`这种方案挺不错，至少能保证业务，但也会增加主库的压力。

3.5.7版本出的`group_replication`多写模式性能不知道如何？架构变动太大，还不敢上。

2019-01-16

作者回复

1. 是这样的，我们说的是，如果非紧急情况下，还是尽量用ghost，在“紧急”的情况下，才这么做；确实是要绕过proxy的，也就是说，这事儿是要负责运维的同学做；
2. pt工具是有这个问题，试一下ghost哈；group_replication多写模式国内我还没有听到国内有公司在生产上大规模用的，如果你有使用经验，分享一下哈

2019-01-16

0



永恒记忆

老师好，有几个问题想请教下，

1. 如果不想有过期读，用等GTID的方案，那么每次查询都要有等GTID的相关操作，增加的这部分对性能有多少影响；
2. 我们用的读写分离proxy不支持等GTID，那是不是自己要在客户端实现这部分逻辑，等于读写分离的架构既用了proxy，又在客户端做了相关策略，感觉这方案更适合有能力自研proxy的公司啊；
3. 感觉目前大多数生产环境还是用的读主库这种方式避免过期读，如果只能用这种方案的话该怎么扩展mysql架构来避免主库压力太大呢。

我们是项目上线很久然后加的读写分离，好多service层代码写的不好，可以读从库的sql被写到了事务中，这样会被proxy转到主库上读，所以导致主库负担了好多读的sql，感觉读写分离不仅对mysql这块要掌握，整体的代码结构上也要有所调整吧。

2019-01-16

0

作者回复

1. 这个等待时间其实就基本上是主备延迟的时间
2. 用了proxy这事情就得proxy做了，就不要客户端做了。没有gtid，可以用倒数第二种方法呀：）
3. 是的，其实“走主库”这种做法还挺多的。我之前看到有的公司的做法，就是直接拆库了。等于一套“一主多从”拆成多套。

2019-01-16

0



HuaMax

课后题。对大表做ddl是一个大事务，等待从库执行，基本就会超时，最后都返回到主库执行，这样的话不如跳过等待从库这一步，但是像老师文中提到需要做好限流。从另一个角度，对于主库的ddl操作，从业务场景去考虑，一般随后到来的查询不会被这个ddl影响，而是对新的业务变更有影响，这样的话，也可以跳过等待从库这一步，直接让从库执行即可。不知道理解是否正确？

2019-01-16

0

作者回复

核心是要处理延迟问题，比如怎么操作可以不会产生延迟

2019-01-16

0



* 晓 *

老师好，如果用MGR或InnoDB cluster方案做读写分离的话可以替代文中提到的方案吗？这两个方案建议在生产中大量使用吗？

2019-01-16

| 作者回复

MGR开始有国内公司在使用了

InnoDB cluster也可以的，但是一般就是平时一写多读，只在主备切换的时候，短暂允许多写

2019-01-16

29 | 如何判断一个数据库是不是出问题了？

2019-01-18 林晓斌



我在第[25](#)和[27](#)篇文章中，和你介绍了主备切换流程。通过这些内容的讲解，你应该已经很清楚了：在一主一备的双M架构里，主备切换只需要把客户端流量切到备库；而在一主多从架构里，主备切换除了要把客户端流量切到备库外，还需要把从库接到新主库上。

主备切换有两种场景，一种是主动切换，一种是被动切换。而其中被动切换，往往是因为主库出问题了，由HA系统发起的。

这也就引出了我们今天要讨论的问题：怎么判断一个主库出问题了？

你一定会说，这很简单啊，连上MySQL，执行个**select 1**就好了。但是**select 1**成功返回了，就表示主库没问题吗？

select 1判断

实际上，**select 1**成功返回，只能说明这个库的进程还在，并不能说明主库没问题。现在，我们来看一下这个场景。

```
set global innodb_thread_concurrency=3;
```

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

```
insert into t values(1,1)
```

session A	session B	session C	session D
select sleep(100) from t;	select sleep(100) from t;	select sleep(100) from t;	
			select 1; (Query OK) select *from t; (blocked)

图1 查询blocked

我们设置`innodb_thread_concurrency`参数的目的是，控制InnoDB的并发线程上限。也就是说，一旦并发线程数达到这个值，InnoDB在接收到新请求的时候，就会进入等待状态，直到有线程退出。

这里，我把`innodb_thread_concurrency`设置成3，表示InnoDB只允许3个线程并行执行。而在我们的例子中，前三个session中的`sleep(100)`，使得这三个语句都处于“执行”状态，以此来模拟大查询。

你看到了，session D里面，`select 1`是能执行成功的，但是查询表t的语句会被堵住。也就是说，如果这时候我们用`select 1`来检测实例是否正常的话，是检测不出问题的。

在InnoDB中，`innodb_thread_concurrency`这个参数的默认值是0，表示不限制并发线程数量。但是，不限制并发线程数肯定是不行的。因为，一个机器的CPU核数有限，线程全冲进来，上下文切换的成本就会太高。

所以，通常情况下，我们建议把`innodb_thread_concurrency`设置为64~128之间的值。这时，你一定会有疑问，并发线程上限数设置为128够干嘛，线上的并发连接数动不动就上千了。

产生这个疑问的原因，是搞混了并发连接和并发查询。

并发连接和并发查询，并不是同一个概念。你在`show processlist`的结果里，看到的几千个连接，指的就是并发连接。而“当前正在执行”的语句，才是我们所说的并发查询。

并发连接数达到几千个影响并不大，就是多占一些内存而已。我们应该关注的是并发查询，因为并发查询太高才是CPU杀手。这也是为什么我们需要设置`innodb_thread_concurrency`参数的原因。

然后，你可能还会想起我们在[第7篇文章](#)中讲到的热点更新和死锁检测的时候，如果把`innodb_thread_concurrency`设置为128的话，那么出现同一行热点更新的问题时，是不是很快就把128消耗完了，这样整个系统是不是就挂了呢？

实际上，在线程进入锁等待以后，并发线程的计数会减一，也就是说等行锁（也包括间隙锁）的线程是不算在128里面的。

MySQL这样设计是非常有意义的。因为，进入锁等待的线程已经不吃CPU了；更重要的是，必须这么设计，才能避免整个系统锁死。

为什么呢？假设处于锁等待的线程也占并发线程的计数，你可以设想一下这个场景：

1. 线程1执行`begin; update t set c=c+1 where id=1`，启动了事务trx1，然后保持这个状态。这时候，线程处于空闲状态，不算在并发线程里面。
2. 线程2到线程129都执行`update t set c=c+1 where id=1`；由于等行锁，进入等待状态。这样就有128个线程处于等待状态；
3. 如果处于锁等待状态的线程计数不减一，InnoDB就会认为线程数用满了，会阻止其他语句进入引擎执行，这样线程1不能提交事务。而另外的128个线程又处于锁等待状态，整个系统就堵住了。

下图2显示的就是这个状态。

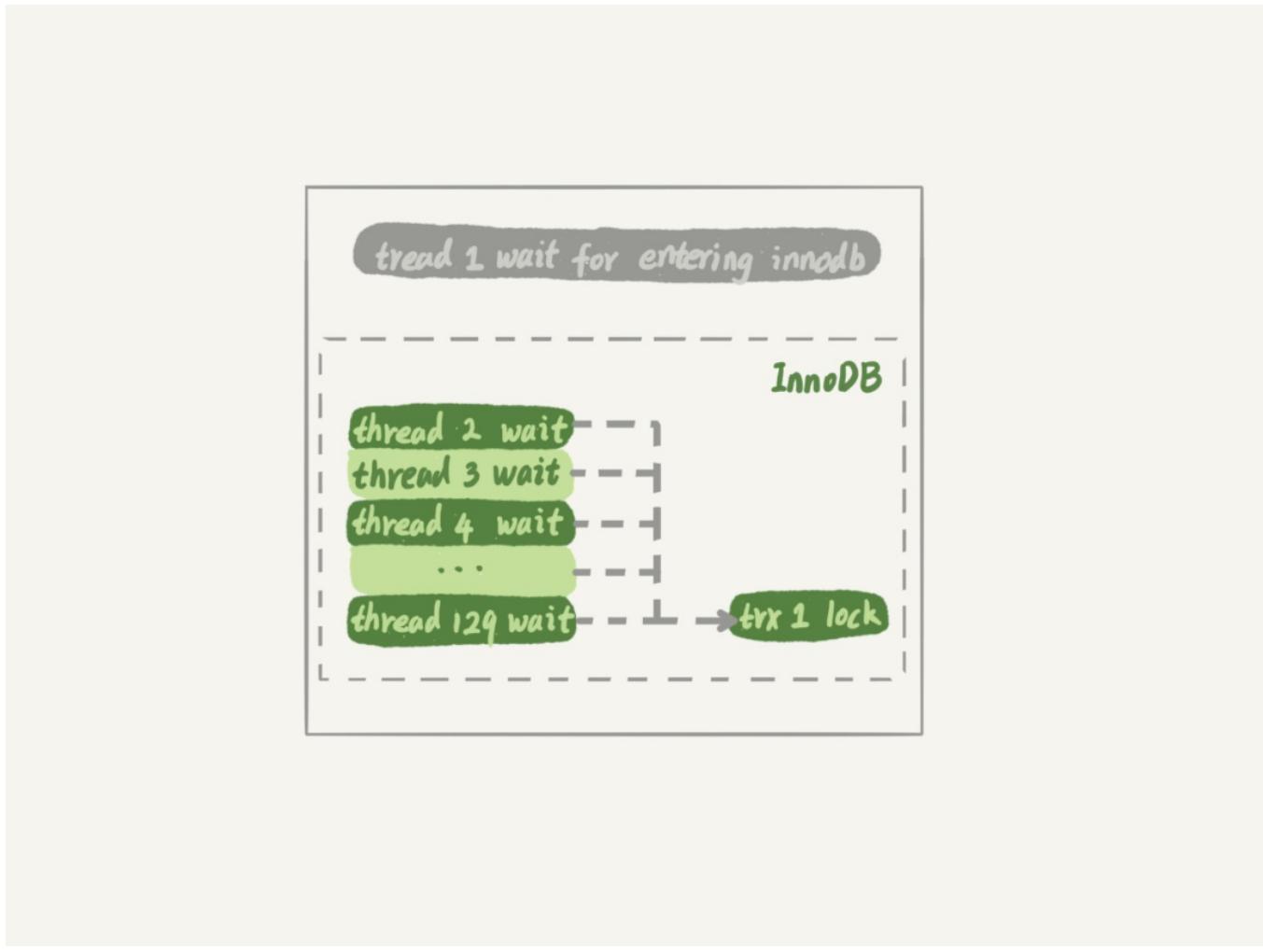


图2 系统锁死状态（假设等行锁的语句占用并发计数）

这时候InnoDB不能响应任何请求，整个系统被锁死。而且，由于所有线程都处于等待状态，此时占用的CPU却是0，而这明显不合理。所以，我们说InnoDB在设计时，遇到进程进入锁等待的情况时，将并发线程的计数减1的设计，是合理而且是必要的。

虽然说等锁的线程不算在并发线程计数里，但如果它在真正地执行查询，就比如我们上面例子中前三个事务中的`select sleep(100) from t`，还是要算进并发线程的计数的。

在这个例子中，同时在执行的语句超过了设置的`innodb_thread_concurrency`的值，这时候系统其实已经不行了，但是通过`select 1`来检测系统，会认为系统还是正常的。

因此，我们使用`select 1`的判断逻辑要修改一下。

查表判断

为了能够检测InnoDB并发线程数过多导致的系统不可用情况，我们需要找一个访问InnoDB的场景。一般的做法是，在系统库（mysql库）里创建一个表，比如命名为`health_check`，里面只放一行数据，然后定期执行：

```
mysql> select * from mysql.health_check;
```

使用这个方法，我们可以检测出由于并发线程过多导致的数据库不可用的情况。

但是，我们马上还会碰到下一个问题，即：空间满了以后，这种方法又会变得不好使。

我们知道，更新事务要写binlog，而一旦binlog所在磁盘的空间占用率达到100%，那么所有的更新语句和事务提交的commit语句就都会被堵住。但是，系统这时候还是可以正常读数据的。

因此，我们还是把这条监控语句再改进一下。接下来，我们就看看把查询语句改成更新语句后的效果。

更新判断

既然要更新，就要放个有意义的字段，常见做法是放一个timestamp字段，用来表示最后一次执行检测的时间。这条更新语句类似于：

```
mysql> update mysql.health_check set t_modified=now();
```

节点可用性的检测都应该包含主库和备库。如果用更新来检测主库的话，那么备库也要进行更新检测。

但，备库的检测也是要写binlog的。由于我们一般会把数据库A和B的主备关系设计为双M结构，所以在备库B上执行的检测命令，也要发回给主库A。

但是，如果主库A和备库B都用相同的更新命令，就可能出现行冲突，也就是可能会导致主备同步停止。所以，现在看来mysql.health_check这个表就不能只有一行数据了。

为了让主备之间的更新不产生冲突，我们可以在mysql.health_check表上存入多行数据，并用A、B的server_id做主键。

```
mysql> CREATE TABLE `health_check` (
    `id` int(11) NOT NULL,
    `t_modified` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;

/* 检测命令 */

insert into mysql.health_check(id, t_modified) values (@@server_id, now()) on duplicate key update t_modified=nc
```

由于MySQL规定了主库和备库的server_id必须不同（否则创建主备关系的时候就会报错），这

样就可以保证主、备库各自的检测命令不会发生冲突。

更新判断是一个相对比较常用的方案了，不过依然存在一些问题。其中，“判定慢”一直是让DBA头疼的问题。

你一定会疑惑，更新语句，如果失败或者超时，就可以发起主备切换了，为什么还会有判定慢的问题呢？

其实，这里涉及到的是服务器IO资源分配的问题。

首先，所有的检测逻辑都需要一个超时时间N。执行一条update语句，超过N秒后还不返回，就认为系统不可用。

你可以设想一个日志盘的IO利用率已经是100%的场景。这时候，整个系统响应非常慢，已经需要做主备切换了。

但是你要知道，IO利用率100%表示系统的IO是在工作的，每个请求都有机会获得IO资源，执行自己的任务。而我们的检测使用的update命令，需要的资源很少，所以可能在拿到IO资源的时候就可以提交成功，并且在超时时间N秒未到达之前就返回给了检测系统。

检测系统一看，update命令没有超时，于是就得到了“系统正常”的结论。

也就是说，这时候在业务系统上正常的SQL语句已经执行得很慢了，但是DBA上去一看，HA系统还在正常工作，并且认为主库现在处于可用状态。

之所以会出现这个现象，根本原因是上面说的所有方法，都是基于外部检测的。外部检测天然有一个问题，就是随机性。

因为，外部检测都需要定时轮询，所以系统可能已经出问题了，但是却需要等到下一个检测发起执行语句的时候，我们才有可能发现问题。而且，如果你的运气不够好的话，可能第一次轮询还不能发现，这就会导致切换慢的问题。

所以，接下来我要再和你介绍一种在MySQL内部发现数据库问题的方法。

内部统计

针对磁盘利用率这个问题，如果MySQL可以告诉我们，内部每一次IO请求的时间，那我们判断数据库是否出问题的方法就可靠得多了。

其实，MySQL 5.6版本以后提供的performance_schema库，就在file_summary_by_event_name表里统计了每次IO请求的时间。

file_summary_by_event_name表里有很多行数据，我们先来看看
event_name='wait/io/file/innodb/innodb_log_file'这一行。

```
mysql> select *  FROM performance_schema.file_summary_by_event_name where event_name = 'wait/io/file/innodb/innodb_log_file'\G
*****1. row ****
EVENT_NAME: wait/io/file/innodb/innodb_log_file
COUNT_STAR: 200192
SUM_TIMER_WAIT: 2495735164992
MIN_TIMER_WAIT: 538080
AVG_TIMER_WAIT: 12446584
MAX_TIMER_WAIT: 3279615840
COUNT_READ: 7
SUM_TIMER_READ: 57395808
MIN_TIMER_READ: 538080
AVG_TIMER_READ: 8199336
MAX_TIMER_READ: 52201056
SUM_NUMBER_OF_BYTES_READ: 70144
COUNT_WRITE: 100093
SUM_TIMER_WRITE: 428658661344
MIN_TIMER_WRITE: 1225728
AVG_TIMER_WRITE: 4282296
MAX_TIMER_WRITE: 26484480
SUM_NUMBER_OF_BYTES_WRITE: 119993344
COUNT_MISC: 100092
SUM_TIMER_MISC: 2067019107840
MIN_TIMER_MISC: 678528
AVG_TIMER_MISC: 20650872
MAX_TIMER_MISC: 3279615840
1 row in set (0.00 sec)
```

图3 performance_schema.file_summary_by_event_name的一行

图中这一行表示统计的是redo log的写入时间，第一列EVENT_NAME 表示统计的类型。

接下来的三组数据，显示的是redo log操作的时间统计。

第一组五列，是所有IO类型的统计。其中，COUNT_STAR是所有IO的总次数，接下来四列是具体的统计项，单位是皮秒；前缀SUM、MIN、AVG、MAX，顾名思义指的就是总和、最小值、平均值和最大值。

第二组六列，是读操作的统计。最后一列SUM_NUMBER_OF_BYTES_READ统计的是，总共从redo log里读了多少个字节。

第三组六列，统计的是写操作。

最后的第四组数据，是对其他类型数据的统计。在redo log里，你可以认为它们就是对fsync的统计。

在performance_schema库的file_summary_by_event_name表里，binlog对应的是event_name = "wait/io/file/sql/binlog"这一行。各个字段的统计逻辑，与redo log的各个字段完全相同。这里，我就不再赘述了。

因为我们每一次操作数据库，performance_schema都需要额外地统计这些信息，所以我们打开这个统计功能是有性能损耗的。

我的测试结果是，如果打开所有的performance_schema项，性能大概会下降10%左右。所以，我建议你只打开自己需要的项进行统计。你可以通过下面的方法打开或者关闭某个具体项的统计。

如果要打开redo log的时间监控，你可以执行这个语句：

```
mysql> update setup_instruments set ENABLED='YES', Timed='YES' where name like '%wait/io/file/innodb/innod
```

假设，现在你已经开启了**redo log**和**binlog**这两个统计信息，那要怎么把这个信息用在实例状态诊断上呢？

很简单，你可以通过**MAX_TIMER**的值来判断数据库是否出问题了。比如，你可以设定阈值，单次**IO**请求时间超过**200毫秒**属于异常，然后使用类似下面这条语句作为检测逻辑。

```
mysql> select event_name,MAX_TIMER_WAIT FROM performance_schema.file_summary_by_event_name where
```

发现异常后，取到你需要的信息，再通过下面这条语句：

```
mysql> truncate table performance_schema.file_summary_by_event_name;
```

把之前的统计信息清空。这样如果后面的监控中，再次出现这个异常，就可以加入监控累积值了。

小结

今天，我和你介绍了检测一个**MySQL**实例健康状态的几种方法，以及各种方法存在的问题和演进的逻辑。

你看完后可能会觉得，**select 1**这样的方法是不是已经被淘汰了呢，但实际上使用非常广泛的**MHA**（**Master High Availability**），默认使用的就是这个方法。

MHA中的另一个可选方法是只做连接，就是“如果连接成功就认为主库没问题”。不过据我所知，选择这个方法的很少。

其实，每个改进的方案，都会增加额外损耗，并不能用“对错”做直接判断，需要你根据业务实际情况去做权衡。

我个人比较倾向的方案，是优先考虑**update**系统表，然后再配合增加检测**performance_schema**的信息。

最后，又到了我们的思考题时间。

今天，我想问你的是：业务系统一般也有高可用的需求，在你开发和维护过的服务中，你是怎么判断服务有没有出问题的呢？

你可以把你用到的方法和分析写在留言区，我会在下一篇文章中选取有趣的方案一起来分享和分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，如果使用GTID等位点的方案做读写分离，在对大表做DDL的时候会怎么样。

假设，这条语句在主库上要执行10分钟，提交后传到备库就要10分钟（典型的大事务）。那么，在主库DDL之后再提交的事务的GTID，去备库查的时候，就会等10分钟才出现。

这样，这个读写分离机制在这10分钟之内都会超时，然后走主库。

这种预期内的操作，应该在业务低峰期的时候，确保主库能够支持所有业务查询，然后把读请求都切到主库，再在主库上做DDL。等备库延迟追上以后，再把读请求切回备库。

通过这个思考题，我主要想让关注的是，大事务对等位点方案的影响。

当然了，使用gh-ost方案来解决这个问题也是不错的选择。

评论区留言点赞板：

@曾剑、@max同学提到的备库先做，再切主库的方法也是可以的。

The image is a promotional graphic for a MySQL course. At the top left is the '极客时间' logo. The main title 'MySQL 实战 45 讲' is displayed prominently in large, bold, dark font. Below it is a subtitle '从原理到实战，丁奇带你搞懂 MySQL' in a smaller, lighter font. To the right of the text is a portrait of the instructor, Lin Xiaobin (丁奇), wearing glasses and a black shirt, with his arms crossed. At the bottom left, there is a section for the author's information: '林晓斌' (Lin Xiaobin) and '网名丁奇 前阿里资深技术专家'. A call-to-action at the bottom right encourages users to upgrade to a new version and share the course with friends for free reading.

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金奖励**。



某、人

6

目前是只有一台服务器来做判断,是否数据库出问题了,就是采用的**update**的方式。如果是主从架构就一条语句,如果是双主的话就是两条**update**语句。但是这种方式有很大的弊端,只有一个进程来判断数据库出问题的话,会出现单点判断的问题。所以后续准备多个单数进程来做判断,如果超过了半数以上的监控进程都认为数据库出问题,才做切换。

老师我有两个问题:

- 1.**innodb_thread_concurrency**的设置是不是应该跟计算机核数成正比,一般是1.5倍-2倍左右?
- 2.怎么之前遇到空间满了,数据库都登不上了,所有的连接都连不上,更不用执行**select**语句了,这个是什么原因啊?

2019-01-20

| 作者回复

1. 虽然理论上是核数的2倍左右最好,但是现在很多人把MySQL创建在虚拟机上,就分1~2个核,我怕那么写,有同学会认为**innodb_thread_concurrency**建议设置成4。。

2. 空间满本身是不会导致连不上的。但是因为空间满,事务无法提交,可能会导致接下来外部事务重试,新重试的业务还是堵在提交阶段,持续累积可能会把连接数用满

2019-01-21



IceGeek17

1

对于使用**GTID**等位点的方案做读写分离,对大表做**DDL**的问题,

有一种做法是先在从库上设置**set_log_bin = off**,在从库上先做**DDL**,完成后做一下主从切换。然后再在之前的主库上同样操作一遍。

但这会有一个问题,当先在从库上做**DDL**(大表**DDL**时间会比较长,比如10分钟),在这段时间内,此时如果读写请求都走主库的话,如果写请求对于**DDL**的改动是有依赖的,那这些写请求在主库就可能会失败;同样此时对于主库上的读请求,也可能会读到“过期”的数据(读请求希望读到**DDL**之后的数据,但此时**DDL**在从库执行,主库上还是**DDL**之前的),老师怎么看这个问题?

2019-01-29

| 作者回复

是这样的,我们说**DDL**,一般是指加减索引,增加字段在最后一列,这种操作...

2019-01-31



Mr.Strive.Z.H.L

1

老师您好:

本章有个疑惑:

“外部检测的时候,主备使用同一条更新语句,造成行冲突,导致主备同步停止”

上面这句话实在想不通。外部检测是只是看更新语句的返回时间, **health_check**表在主库备库都有,为啥会造成行冲突?为啥会导致主备同步停止?即使是相同的binlog,也没啥影响呀。

2019-01-22

| 作者回复

比如两个表刚开始都是空表，

然后第一个语句执行

```
insert into mysql.health_check(id, t_modified) values (1, now()) on duplicate key update t_modified=now();
```

就会两边各写入一个insert语句的binlog日志，传到对面就导致同步停止了

2019-01-22



慧鑫coming

1

老师，文中提到的“但是，如果主库 A 和备库 B 都用相同的更新命令，就可能出现行冲突，也就是可能会导致主备同步停止。”，这个能展开说一下吗，这个行冲突指什么？它们会都更新各自检测表的同一字段我觉得会带来不准确的问题，怎么导致主从同步停止了呢？

2019-01-22

| 作者回复

好问题

比如两个表刚开始都是空表，

然后第一个语句执行

```
insert into mysql.health_check(id, t_modified) values (1, now()) on duplicate key update t_modified=now();
```

就会两边各写入一个insert语句的binlog日志，传到对面就导致同步停止了

2019-01-22



heat nan

1

老师，一直有个疑问，想咨询下。innodb buffer 会缓存表的数据页和索引页。现在我想知道如何确认一个查询的行已经被缓存在内存中了。我想了一下，第一种方法是直接去内存中遍历这个表相关的数据页。这样的话，因为内存中的页可能是分散的，可能不构成一个完成的索引结构，可能不能利用b+树叶子节点的路由功能。这里有点模糊，希望老师有空可以解释一下

2019-01-19

| 作者回复

“因为内存中的页可能是分散的，可能不构成一个完成的索引结构，可能不能利用b+树叶子节点的路由功能。”

这里不对哈

放在内存里是b+树组织的，可以利用b+树叶子节点的路由功能的

2019-01-19



老杨同志

1

现在很多公司都是使用**dubbo**或者类似**dubbo**的**rpc**调用。说说我对**dubbo**的理解

dubbo 存活检测感觉分为下面三个层面

服务端与注册中心的链接状态

通常注册中心是**zookeeper**, 服务端注册临时节点, 客户端注册这个节点的**watch**事件, 一但服务端失联,

客户端将把该服务从自己可用服务列表中移除。（一个服务通常有多个提供者, 只是把失联的提供者移除）。

zookeeper是通过心跳发现服务提供者失联的, 心跳实际上就是以固定的频率（比如每秒）发送检测的数据包;

客户端与注册中心的链接状态

客户端与**zookeeper**失联, 会暂时使用自己缓存的服务提供者列表。如果每个提供者多次调不通, 把它移除。

客户端与服务单的链接状态

服务端提供类似于**echo**的方法, 客户定时调用。部分返回正常, 认为服务处于亚健康状态, 如果超过阀值, 会被降级

从服务提供者列表移除。被移除的方法可能会在超过一定时间后, 拿回来重试, 可以恢复正常服务, 也可能继续降级。

2019-01-18

| 作者回复

很好的实践分享。

是不是还有配套一些服务的**RT**时间的报告？

毕竟**echo**是一个比较轻量的调用, 正确率可能比实际业务调用的正确率高

2019-01-20



强哥

1

1.基础监控, 包括硬盘, CPU, 网络, 内存等。

2.服务监控, 包括jvm, 服务端口, 接入上下游服务的超时监控等。

3.业务监控, 主要是监控业务的流程是否出现问题。

2019-01-18

| 作者回复

[], 这里的“超时监控”, 是怎么得到的?

是单独有命令检测, 还是去看业务请求的返回时间?

2019-01-18



长杰

1

老师请教一个问题, 在**gtid**模式下, 对于大的**ddl**操作, 采用在备库执行**sql_log_bin=0**的方式先执行, 然后再切换主备的方式在主库再执行, 这种情况下, **ddl**操作是不记录**binlog**的, 不知道对**gtid**的计数有什么影响, 是按顺序递增还是会跳过这个序列号?

另外补充一下有些**dl**操作是不适合这个主备切换的方式, 比如**drop**一个列, 如果先在备库执行就可能导致主备同步异常。这个场景适合**osc**方式或把读请求切到主库, 先在主库执行这两种方案。

2019-01-18

作者回复

如果`set sql_log_bin=0`, 就不记录binlog, 就不会给这个事务分配gtid。

你说得对, `drop`列是很麻烦的, 尽量不做。毕竟业务代码直接无视这个列就好了。。

2019-01-18



Mr.Strive.Z.H.L

0

老师您好:

关于主备同步停止的问题, 看了您的回复。

我是这么理解的:

```
insert into mysql.health_check(id, t_modified) values (1, now()) on duplicate key update t_modified=now();
```

按照您说的场景, 主备分别执行这句话后, 复制给彼此。

如果单单看这句话, 就算是主库执行备库复制过来的这句话, 也不会出现异常呀。(因为如果主键冲突就会更新时间)

但是这种场景会导致主备同步停止, 所以实际上主库在应用备库这句话的binlog的时候, 发现主键冲突, 自然就会报错。

不知道是不是这样, 因为如果单单看这句sql, 即使主键冲突也没关系呀?

2019-01-22

作者回复

啊 主键冲突为啥没关系?

是这样的, 这两个语句如果同时执行, 那么在主库和备库上就都是“`insert`行为”

写到binlog里面就都是`Write rows event`

这个冲突就会导致主备同步停止哦

2019-01-23



一大只

0

老师, 我想问下, 我的ECS上是8核CPU, 只跑一个MySQL实例, 那`innodb_thread_concurrency`如果设成2倍, 那就是16哈。看并发查询的数量, 是不是关注`Threads_running`是否超过`innodb_thread_concurrency`就可以了。

2019-01-21

作者回复

`Thread running`是包含“锁等待”状态的线程的,

超过点也没事

2019-01-22



小橙橙

0

老师, 我工作中遇到一个奇怪的问题, java客户端执行查询语句报错: `ResultSet is from UPDATE. No Data.`。用navicat执行相同语句, 很快就查询结束, 但是没有结果显示。请问可能什么问题造成的呢?

2019-01-18

| 作者回复

? 这两个不是一致的吗

意思就是你要**update**的语句找不到呀

你把**update**改成**select**, 先确定一下是不是能看到你要更新的数据（根据你这个描述，应该是没有）

2019-01-18



悟空

0

可以大致从**DB**监控图上判断业务有没有问题：

QPS/连接数/慢查询/查询响应时间(**query_response_time**插件)等.....

老师请教一个问题：

物理机器是**128G**内存, **DB**实例数据量是**1.2T**, 磁盘是**pcie ssd**

业务查询场景是简单的**select * from table where id in (1,2,3,...);**

实例**QPS**在**1000**以下时,数据库看上去一切正常

当**QPS**大于**2000+**时, **%util**持续**90+**, **r/s**持续**2W**左右, **rMB/s**持续**600+**, 伴随着连接数/慢查询等报警

这个时候这个数据库实例可以说是出问题了吧, 这类问题该怎么排查根因呢?

是由于**buffer pool**与磁盘大量换入换出冷数据导致的吗, 有相关的状态值监控项可以查吗?

innodb buffer pool是**mysql**很重要的一个模块, 老师后面有单独的章节来解惑吗, 期待！！

2019-01-18

| 作者回复

有的, 敬请期待

不过**buffer pool**内部细节很多, 只能挑大家使用的时候, 可能会用到的知识点来讲哈

2019-01-18



One day

0

作为一个开发我也很想了解一下我们自己生产库上的监控情况, 接触到最多的就是**Datasource**, 以及**user, password, port** (基本上是基于连接那种级别, 最多就是加锁), 等等参数, 大部分都是基于业务开发。站在个人层面或者业务开发层面 (很少能接触到**DBA**, 以及看到**DBA**是怎么设置这些参数情况, 除非库挂掉了就会和**DBA**一起看这些) 怎么去修改和观看以及使用这些参数鸭

2019-01-18

| 作者回复

有**DBA**就不要自己去修改线上的参数啦

如果说观察, 一个比较好的管控系统, 是能够让你看到这些值的

如果没有, 就让**dba**给你一份线上的**my.cnf**的配置, 然后你在测试环境自己用这个配置启动实例

来观察

2019-01-18



Ryoma

0

现在的服务中只加了一个**healthCheck**的接口，和MySQL中使用**select**判断比较类似。当服务依赖的MySQL及Redis等第三方资源发生问题时，还是不能有效的判断

2019-01-18



爸爸回来了

0

之前也用**select 1**。后来发现，硬盘意外塞满时，本地链接很有可能超时导致判断失败。
想问问老师，**mysqladmin ping**这个机制用来判断如何？

2019-01-18

| 作者回复

跟**select 1**属于一类

2019-01-18

30 | 答疑文章（二）：用动态的观点看加锁

2019-01-21 林晓斌



在第[20](#)和[21](#)篇文章中，我和你介绍了InnoDB的间隙锁、next-key lock，以及加锁规则。在这两篇文章的评论区，出现了很多高质量的留言。我觉得通过分析这些问题，可以帮助你加深对加锁规则的理解。

所以，我就从中挑选了几个有代表性的问题，构成了今天这篇答疑文章的主题，即：用动态的观点看加锁。

为了方便你理解，我们再一起复习一下加锁规则。这个规则中，包含了两个“原则”、两个“优化”和一个“bug”：

- 原则1：加锁的基本单位是next-key lock。希望你还记得，next-key lock是前开后闭区间。
- 原则2：查找过程中访问到的对象才会加锁。
- 优化1：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。
- 优化2：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。
- 一个bug：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。

接下来，我们的讨论还是基于下面这个表t：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `c` (`c`)
) ENGINE=InnoDB;

insert into t values(0,0,0),(5,5,5),
(10,10,10),(15,15,15),(20,20,20),(25,25,25);
```

不等号条件里的等值查询

有同学对“等值查询”提出了疑问：等值查询和“遍历”有什么区别？为什么我们文章的例子里面，**where**条件是不等号，这个过程里也有等值查询？

我们一起来看下这个例子，分析一下这条查询语句的加锁范围：

```
begin;
select * from t where id>9 and id<12 order by id desc for update;
```

利用上面的加锁规则，我们知道这个语句的加锁范围是主键索引上的 $(0,5]$ 、 $(5,10]$ 和 $(10, 15)$ 。也就是说， $id=15$ 这一行，并没有被加上行锁。为什么呢？

我们说加锁单位是**next-key lock**，都是前开后闭区间，但是这里用到了优化2，即索引上的等值查询，向右遍历的时候 $id=15$ 不满足条件，所以**next-key lock** 退化为了间隙锁 $(10, 15)$ 。

但是，我们的查询语句中**where** 条件是大于号和小于号，这里的“等值查询”又是从哪里来的呢？

要知道，加锁动作是发生在语句执行过程中的，所以你在分析加锁行为的时候，要从索引上的数据结构开始。这里，我再把这个过程拆解一下。

如图1所示，是这个表的索引**id**的示意图。

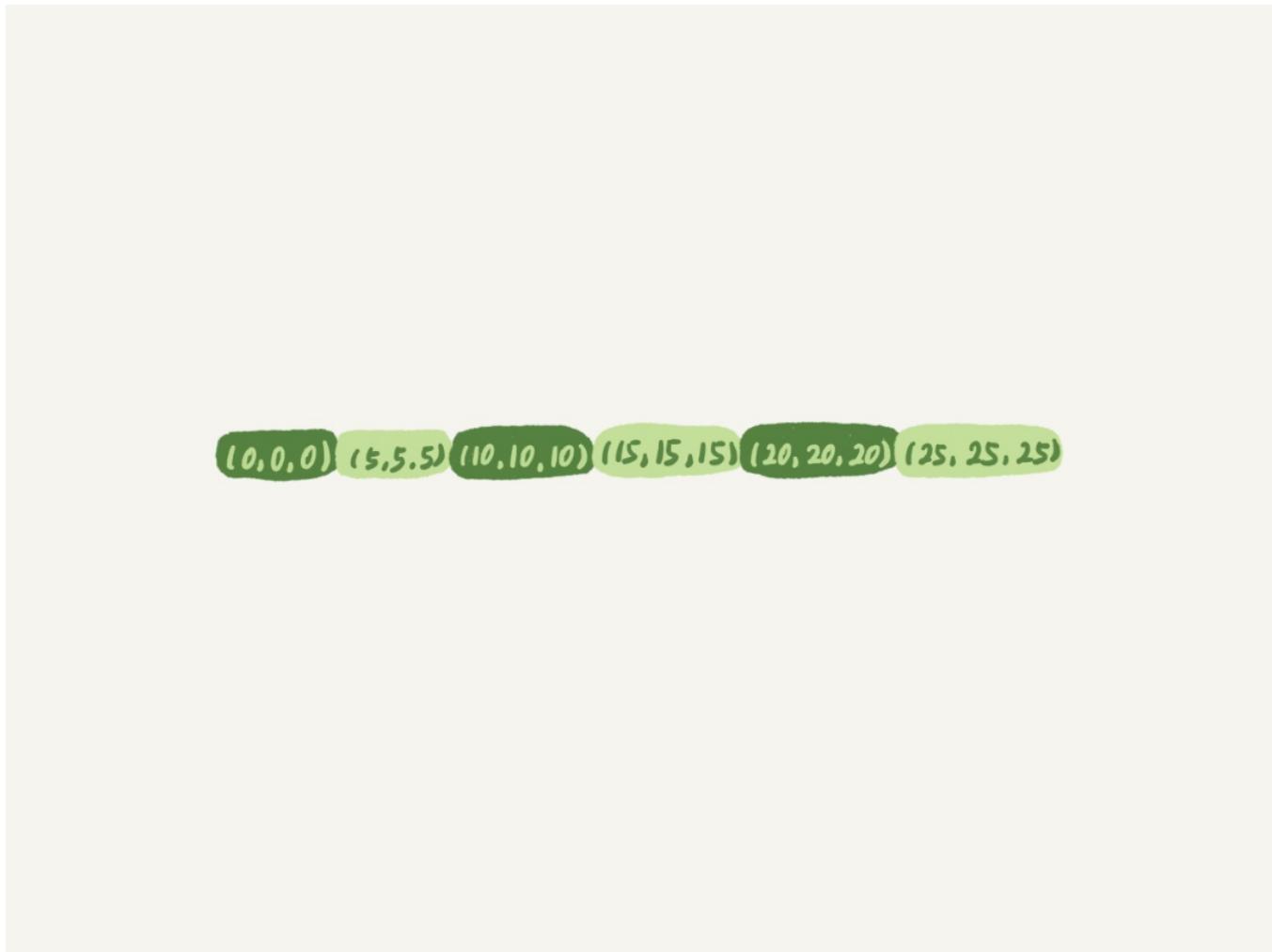


图1 索引id示意图

- 首先这个查询语句的语义是**order by id desc**, 要拿到满足条件的所有行, 优化器必须先找到“第一个**id<12**的值”。
- 这个过程是通过索引树的搜索过程得到的, 在引擎内部, 其实是要找到**id=12**的这个值, 只是最终没找到, 但找到了**(10,15)**这个间隙。
- 然后向左遍历, 在遍历过程中, 就不是等值查询了, 会扫描到**id=5**这一行, 所以会加一个**next-key lock (0,5]**。

也就是说, 在执行过程中, 通过树搜索的方式定位记录的时候, 用的是“等值查询”的方法。

等值查询的过程

与上面这个例子对应的, 是@发条橙子同学提出的问题: 下面这个语句的加锁范围是什么?

```
begin;  
select id from t where c in(5,20,10) lock in share mode;
```

这条查询语句里用的是**in**, 我们先来看这条语句的**explain**结果。

mysql> explain select id from t where c in(5,20,10) lock in share mode;											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t	NULL	range	c	c	5	NULL	3	100.00	Using where; Using index

图2 in语句的explain结果

可以看到，这条in语句使用了索引c并且rows=3，说明这三个值都是通过B+树搜索定位的。

在查找c=5的时候，先锁住了[0,5]。但是因为c不是唯一索引，为了确认还有没有别的记录c=5，就要向右遍历，找到c=10才确认没有了，这个过程满足优化2，所以加了间隙锁(5,10)。

同样的，执行c=10这个逻辑的时候，加锁的范围是(5,10] 和 (10,15)；执行c=20这个逻辑的时候，加锁的范围是(15,20] 和 (20,25)。

通过这个分析，我们可以知道，这条语句在索引c上加的三个记录锁的顺序是：先加c=5的记录锁，再加c=10的记录锁，最后加c=20的记录锁。

你可能会说，这个加锁范围，不就是从(5,25)中去掉c=15的行锁吗？为什么这么麻烦地分段说呢？

因为我要跟你强调这个过程：这些锁是“在执行过程中一个一个加的”，而不是一次性加上去的。

理解了这个加锁过程之后，我们就可以来分析下面例子中的死锁问题了。

如果同时有另外一个语句，是这么写的：

```
select id from t where c in(5,20,10) order by c desc for update;
```

此时的加锁范围，又是什么呢？

我们现在都知道间隙锁是不互锁的，但是这两条语句都会在索引c上的c=5、10、20这三行记录上加记录锁。

这里你需要注意一下，由于语句里面是order by c desc，这三个记录锁的加锁顺序，是先锁c=20，然后c=10，最后是c=5。

也就是说，这两条语句要加锁相同的资源，但是加锁顺序相反。当这两条语句并发执行的时候，就可能出现死锁。

关于死锁的信息，MySQL只保留了最后一个死锁的现场，但这个现场还是不完备的。

有同学在评论区留言到，希望我能展开一下怎么看死锁。现在，我就来简单分析一下上面这个例子的死锁现场。

怎么看死锁？

图3是在出现死锁后，执行`show engine innodb status`命令得到的部分输出。这个命令会输出很多信息，有一节**LATESTDETECTED DEADLOCK**，就是记录的最后一次死锁信息。

```
2019-01-09 19:21:11 0x7feb98d47700
*** (1) TRANSACTION:
TRANSACTION 422127109356256, ACTIVE 0 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1136, 3 row lock(s)
MySQL thread id 98, OS thread handle 140649857836800, query id 119190 localhost 127.0.0.1 root Sending data
select id from t where c in(5,20,10) lock in share mode
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 24 page no 4 n bits 80 index c of table `test`.`t` trx id 422127109356256 lock mode S waiting
Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 0000000a; asc      ;;
 1: len 4; hex 0000000a; asc      ;;

*** (2) TRANSACTION:
TRANSACTION 1315, ACTIVE 0 sec starting index read
mysql tables in use 1, locked 1
5 lock struct(s), heap size 1136, 7 row lock(s)
MySQL thread id 99, OS thread handle 140649858103040, query id 119189 localhost 127.0.0.1 root Sending data
select id from t where c in(5,20,10) order by c desc for update
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 24 page no 4 n bits 80 index c of table `test`.`t` trx id 1315 lock_mode X
Record lock, heap no 4 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 0000000a; asc      ;;
 1: len 4; hex 0000000a; asc      ;;

Record lock, heap no 6 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 00000014; asc      ;;
 1: len 4; hex 00000014; asc      ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 24 page no 4 n bits 80 index c of table `test`.`t` trx id 1315 lock_mode X waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 00000005; asc      ;;
 1: len 4; hex 00000005; asc      ;;

*** WE ROLL BACK TRANSACTION (1)
```

图3 死锁现场

我们来看看这图中的几个关键信息。

1. 这个结果分成三部分：
 - (1) TRANSACTION, 是第一个事务的信息；
 - (2) TRANSACTION, 是第二个事务的信息；
 - WE ROLL BACK TRANSACTION (1), 是最终的处理结果，表示回滚了第一个事务。
2. 第一个事务的信息中：
 - WAITING FOR THIS LOCK TO BE GRANTED, 表示的是这个事务在等待的锁信息；
 - index c of table `test`.`t`，说明在等的是表t的索引c上面的锁；
 - lock mode S waiting 表示这个语句要自己加一个读锁，当前的状态是等待中；
 - Record lock说明这是一个记录锁；
 - n_fields 2表示这个记录是两列，也就是字段c和主键字段id；
 - 0: len 4; hex 0000000a; asc ;;是第一个字段，也就是c。值是十六进制a，也就是10；
 - 1: len 4; hex 0000000a; asc ;;是第二个字段，也就是主键id，值也是10；
 - 这两行里面的asc表示的是，接下来要打印出值里面的“可打印字符”，但10不是可打印

字符，因此就显示空格。

- 第一个事务信息就只显示出了等锁的状态，在等待($c=10, id=10$)这一行的锁。
- 当然你是知道的，既然出现死锁了，就表示这个事务也占有别的锁，但是没有显示出来。别着急，我们从第二个事务的信息中推导出来。

3. 第二个事务显示的信息要多一些：

- “**HOLDS THE LOCK(S)**”用来显示这个事务持有哪些锁；
- `index c of table `test`.`t`` 表示锁是在表t的索引c上；
- `hex 0000000a`和`hex 00000014`表示这个事务持有 $c=10$ 和 $c=20$ 这两个记录锁；
- **WAITING FOR THIS LOCK TO BE GRANTED**，表示在等($c=5, id=5$)这个记录锁。

从上面这些信息中，我们就知道：

1. “**lock in share mode**”的这条语句，持有 $c=5$ 的记录锁，在等 $c=10$ 的锁；
2. “**for update**”这个语句，持有 $c=20$ 和 $c=10$ 的记录锁，在等 $c=5$ 的记录锁。

因此导致了死锁。这里，我们可以得到两个结论：

1. 由于锁是一个个加的，要避免死锁，对同一组资源，要按照尽量相同的顺序访问；
2. 在发生死锁的时刻，**for update** 这条语句占有的资源更多，回滚成本更大，所以**InnoDB**选择了回滚成本更小的**lock in share mode**语句，来回滚。

怎么看锁等待？

看完死锁，我们再来看一个锁等待的例子。

在第21篇文章的评论区，`@Geek_9ca34e` 同学做了一个有趣验证，我把复现步骤列出来：

session A	session B
<code>begin;</code> <code>select * from t where id>10 and id<=15 for update;</code>	
	<code>delete from t where id=10;</code> (Query OK)

图4 `delete`导致间隙变化

可以看到，由于**session A**并没有锁住 $c=10$ 这个记录，所以**session B**删除 $id=10$ 这一行是可行的。但是之后，**session B**再想`insert id=10`这一行回去就不行了。

现在我们一起看一下此时`show engine innodb status`的结果，看看能不能给我们一些提示。锁信息是在这个命令输出结果的TRANSACTIONS这一节。你可以在文稿中看到这张图片

```
--TRANSACTION 1311, ACTIVE 4 sec inserting
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 11, OS thread handle 140504341464832, query id 20700 localhost 127.0.0.1 root update
insert into t values(10,10,10)
----- TRX HAS BEEN WAITING 4 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 24 page no 3 n bits 80 index PRIMARY of table `test`.`t` trx id 1311 lock_mode X locks gap before rec insert intention waiting
Record lock, heap no 5 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
0: len 4; hex 0000000f; asc   ;;
1: len 6; hex 000000000513; asc   ;;
2: len 7; hex b0000001250134; asc % 4;;
3: len 4; hex 0000000f; asc   ;;
4: len 4; hex 8000000f; asc   ;;


```

图 5 锁等待信息

我们来看几个关键信息。

1. `index PRIMARY of table `test`.`t``，表示这个语句被锁住是因为表t主键上的某个锁。
2. `lock_mode X locks gap before rec insert intention waiting` 这里有几个信息：
 - `insert intention` 表示当前线程准备插入一个记录，这是一个插入意向锁。为了便于理解，你可以认为它就是这个插入动作本身。
 - `gap before rec` 表示这是一个间隙锁，而不是记录锁。
3. 那么这个`gap`是在哪个记录之前的呢？接下来的0~4这5行的内容就是这个记录的信息。
4. `n_fields 5`也表示了，这一个记录有5列：
 - 0: `len 4; hex 0000000f; asc ;` 第一列是主键id字段，十六进制f就是`id=15`。所以，这时我们就知道了，这个间隙就是`id=15`之前的，因为`id=10`已经不存在了，它表示的就是`(5,15)`。
 - 1: `len 6; hex 000000000513; asc ;` 第二列是长度为6字节的事务id，表示最后修改这一行的是`trx id`为1299的事务。
 - 2: `len 7; hex b0000001250134; asc % 4;;` 第三列长度为7字节的回滚段信息。可以看到，这里的`acs`后面有显示内容(%和4)，这是因为刚好这个字节是可打印字符。
 - 后面两列是c和d的值，都是15。

因此，我们就知道了，由于`delete`操作把`id=10`这一行删掉了，原来的两个间隙`(5,10)、(10,15)`变成了一个`(5,15)`。

说到这里，你可以联合起来再思考一下这两个现象之间的关联：

1. session A执行完`select`语句后，什么都没做，但它加锁的范围突然“变大”了；
2. 第21篇文章的课后思考题，当我们执行`select * from t where c>=15 and c<=20 order by c desc lock in share mode;` 向左扫描到`c=10`的时候，要把`(5, 10]`锁起来。

也就是说，所谓“间隙”，其实根本就是由“这个间隙右边的那个记录”定义的。

update的例子

看过了insert和delete的加锁例子，我们再来看一个update语句的案例。在留言区中@信信 同学做了这个试验：

sesison A	session B
begin; select c from t where c >5 lock in share mode;	
	update t set c = 1 where c = 5; (Query OK) update t set c = 5 where c = 1; (blocked)

图 6 update 的例子

你可以自己分析一下，session A的加锁范围是索引c上的(5,10]、(10,15]、(15,20]、(20,25]和(25,supremum]。

注意：根据c>5查到的第一个记录是c=10，因此不会加(0,5]这个next-key lock。

之后session B的第一个update语句，要把c=5改成c=1，你可以理解为两步：

1. 插入(c=1, id=5)这个记录；
2. 删除(c=5, id=5)这个记录。

按照我们上一节说的，索引c上(5,10)间隙是由这个间隙右边的记录，也就是c=10定义的。所以通过这个操作，session A的加锁范围变成了图7所示的样子：

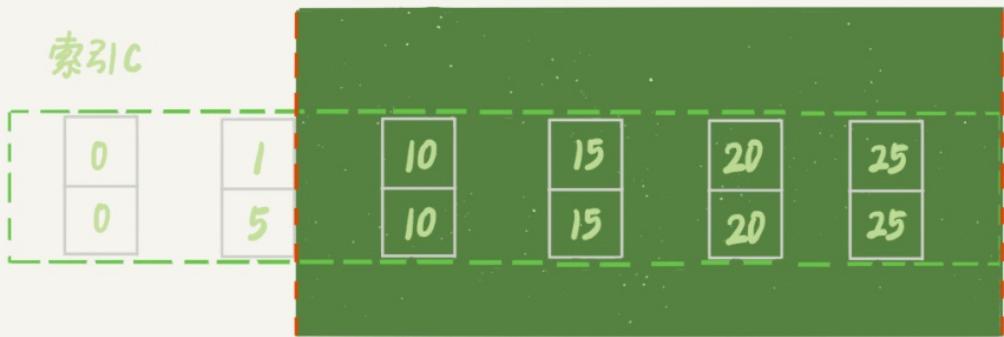


图 7 session B修改后， session A的加锁范围

好，接下来session B要执行 `update t set c = 5 where c = 1`这个语句了，一样地可以拆成两步：

1. 插入($c=5, id=5$)这个记录；
2. 删 除($c=1, id=5$)这个记录。

第一步试图在已经加了间隙锁的(1,10)中插入数据，所以就被堵住了。

小结

今天这篇文章，我用前面[第20](#)和[第21](#)篇文章评论区的几个问题，再次跟你复习了加锁规则。并且，我和你重点说明了，分析加锁范围时，一定要配合语句执行逻辑来进行。

在我看来，每个想认真了解MySQL原理的同学，应该都要能够做到：通过`explain`的结果，就能够脑补出一个SQL语句的执行流程。达到这样的程度，才算是对索引组织表、索引、锁的概念有了比较清晰的认识。你同样也可以用这个方法，来验证自己对这些知识点的掌握程度。

在分析这些加锁规则的过程中，我也顺便跟你介绍了怎么看`show engine innodb status`输出结果中的事务信息和死锁信息，希望这些内容对你以后分析现场能有所帮助。

老规矩，即便是答疑文章，我也还是要留一个课后问题给你的。

上面我们提到一个很重要的点：所谓“间隙”，其实根本就是由“这个间隙右边的那个记录”定义的。

那么，一个空表有间隙吗？这个间隙是由谁定义的？你怎么验证这个结论呢？

你可以把你关于分析和验证方法写在留言区，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章最后留给的问题，是分享一下你关于业务监控的处理经验。

在这篇文章的评论区，很多同学都分享了不错的经验。这里，我就选择几个比较典型的留言，和你分享吧：

- @老杨同志 回答得很详细。他的主要思路就是关于服务状态和服务质量的监控。其中，服务状态的监控，一般都可以用外部系统来实现；而服务的质量的监控，就要通过接口的响应时间来统计。
- @Ryoma 同学，提到服务中使用了**healthCheck**来检测，其实跟我们文中提到的**select 1**的模式类似。
- @强哥 同学，按照监控的对象，将监控分成了基础监控、服务监控和业务监控，并分享了每种监控需要关注的对象。

这些都是很好的经验，你也可以根据具体的业务场景借鉴适合自己的方案。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Lin Xiaobin, a man with short dark hair and glasses, wearing a black button-down shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45 讲' in large, bold, dark font, with the subtitle '从原理到实战，丁奇带你搞懂 MySQL' below it. At the top left is the '极客时间' logo. Below the title, there's a section with the author's name '林晓斌' and his alias '网名丁奇 前阿里资深技术专家'. At the bottom, there's a call-to-action: '新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。'.

精选留言



令狐少侠

2

有个问题想确认下，在死锁日志里，`lock_mode X waiting`是间隙锁+行锁，`lock_mode X locks rec but not gap`这种加`but not gap`才是行锁？

老师你后面能说下`group by`的原理吗，我看目录里面没有

2019-01-22

| 作者回复

对，好问题

`lock_mode X waiting`表示next-key lock；

`lock_mode X locks rec but not gap`是只有行锁；

还有一种“`locks gap before rec`”，就是只有间隙锁；

2019-01-23



Ryoma

2

删除数据，导致锁扩大的描述：“因此，我们就知道了，由于`delete`操作把`id=10`这一行删掉了，原来的两个间隙`(5,10)`、`(10,15)`变成了一个`(5,15)`。”

我觉得这个提到的`(5, 10)`和`(10, 15)`两个间隙会让人有点误解，实际上在删除之前间隙锁只有一个`(10, 15)`，删除了数据之后，导致间隙锁左侧扩张成了`5`，间隙锁成为了`(5, 15)`。

2019-01-22

| 作者回复

嗯 所以我这里特别小心地没有写“锁”这个字。

间隙`(5,10)`、`(10,15)`是客观存在的。

你提得也很对，“锁”是执行过程中才加的，是一个动态的概念。

这个问题也能够让大家更了解我们标题的意思，置顶了哈『

2019-01-22



』

1

老师好：

`select * from t where c>=15 and c<=20 order by c desc for update;`

为什么这种`c=20`就是用来查数据的就不是向右遍历

`select * from t where c>=15 and c<=20`这种就是向右遍历

怎么去判断合适是查找数据，何时又是遍历呢，是因为第一个有`order by desc`，然后反向向左遍历了吗？所以只需要`[20,25]`来判断已经是最后一个`20`就可以了是吧

2019-01-22

| 作者回复

索引搜索就是“找到第一个值，然后向左或向右遍历”，

`order by desc`就是要用最大的值来找第一个；

`order by`就是要用做小的值来找第一个；

“所以只需要`[20,25)`来判断已经是最后一个`20`就可以了是吧”，

你描述的意思是对的，但是在MySQL里面不建议写这样的前闭后开区间哈，容易造成误解。

可以描述为：

“取第一个`id=20`后，向右遍历`(25,25)`这个间隙”^_^

2019-01-22



老杨同志

1

先说结论：空表锁`(-supernum, supernum]`,老师提到过mysql的正无穷是`supernum`，在没有数据的情况下，`next-key lock`应该是`supernum`前面的间隙加`supernum`的行锁。但是前开后闭的区间，前面的值是什么我也不知道，就写了一个`-supernum`。

稍微验证一下

session 1)

```
begin;  
select * from t where id>9 for update;
```

session 2)

```
begin;  
insert into t values(0,0,0),(5,5,5);
```

(block)

2019-01-21

| 作者回复

赞

`show engine innodb status` 有惊喜

2019-01-21



Long

0

感觉这篇文章以及前面加锁的文章，提升了自己的认知。还有，谢谢老师讲解了日志的对应细节.....还愿了

2019-01-28

| 作者回复

||

2019-01-28



滔滔

0

老师，有个疑问，`select * from t where c>=15 and c<=20 order by c desc lock in share mode;` 向左扫描到`c=10`的时候，为什么要把`(5, 10]`锁起来？不锁也不会出现幻读或者逻辑上的不一致吧

2019-01-23

| 作者回复

会加锁，`insert into t values (6,6,6)` 被堵住了

2019-01-23

尘封

回 U

老师，咨询个问题，本来想在后面分区表的文章问，发现大纲里没有分区表这一讲。

1, timestamp类型为什么不支持分区？

2, 前面的文章讲过分区不要太多，这个多了会怎么样？比如一个表一千多个分区

谢谢

2019-01-23

| 作者回复

会讲的哈~

新春快乐~

2019-02-04



长杰

0

老师，还是`select * from t where c>=15 and c<=20 order by c desc in share mode`与`select * from t where id>10 and id<=15 for update`的问题，为何`select * from t where id>10 and id<=15 for update`不能解释为：根据`id=15`来查数据，加锁`(15, 20]`的时候，可以使用优化2，这个等值查询是根据什么规则来定的？如果`select * from t where id>10 and id<=15 for update`加上`order by id desc`是否可以按照`id=15`等值查询，利用优化2？多谢指教。

2019-01-22

| 作者回复

1. 代码实现上，传入的就是`id>10`里面的这个`10`

2. 可以的，不过因为`id`是主键，而且`id=15`这一行存在，我觉得用优化1解释更好哦

2019-01-23



堕落天使

0

老师，您好：

我执行“`explain select id from t where c in(5,20,10) lock in share mode;`”时，显示的`rows`对应的值是4。为什么啊？

我的mysql版本是：5.7.23-0ubuntu0.16.04.1，具体sql语句如下：

`mysql> select * from t;`

`+----+-----+-----+`

`| id | c | d |`

`+----+-----+-----+`

`| 0 | 0 | 0 |`

`| 5 | 5 | 5 |`

`| 10 | 10 | 10 |`

`| 15 | 15 | 15 |`

`| 20 | 20 | 20 |`

`| 25 | 25 | 25 |`

`| 30 | 10 | 30 |`

`+----+-----+-----+`

`7 rows in set (0.00 sec)`

`mysql> explain select id from t where c in(5,20,10) lock in share mode;`

```
+-----+-----+-----+-----+-----+-----+-----+
-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered |
| Extra |
+-----+-----+-----+-----+-----+-----+-----+
-----+
| 1 | SIMPLE | t | NULL | range | c | c | 5 | NULL | 4 | 100.00 | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+
-----+
1 row in set, 1 warning (0.00 sec)
```

2019-01-22

作者回复

你这个例子里面有两行c=10

2019-01-23



Ivan

0

Jan 17 23:52:27 prod-mysql-01 kernel: [pid] uid tgid total_vm rss cpu oom_adj oom_score_adj
name
Jan 17 23:52:27 prod-mysql-01 kernel: [125254] 0 125254 27087 5 0 0 0 mysqld_safe
Jan 17 23:52:27 prod-mysql-01 kernel: [126004] 498 126004 24974389 22439356 5 0 0 mysqld
Jan 17 23:52:27 prod-mysql-01 kernel: [5733] 0 5733 7606586 6077037 7 0 0 mysql

系统日志-----

老师你好，请教一个问题，我在mysql服务器上本地登录，执行了一个SQL（`select b.id,b.status from rb_bak b where id not in (select id from rb)`);该语句问了找不同数据，rb和rb_bak 数据量均为500万左右），SQL很慢，30分钟也没结果；

在SQL语句执行期间，发生了OOM，mysql服务被kill。查看系统日志发现 mysqld 占用内存基本没有变，但是本机连接mysql的客户端进程（5733）却占用了内存近20G，这很让人费解，SQL没有执行完，客户端怎么会占用这么多内存？

用其他SQL查询查询不同数据，也就十几条数据，更不可能占用这么多内存呀。还请老师帮忙分析一下，谢谢。

2019-01-22

作者回复

好问题，第33篇会说到哈

你可以在mysql客户端参数增加 `--quick` 再试试

2019-01-23



PengfeiWang

0

老师，您好：

对文中以下语句感到有困惑：

我们说加锁单位是 `next-key lock`，都是前开后闭区，但是这里用到了优化 2，即索引上的等值查询，向右遍历的时候`id=15`不满足条件，所以 `next-key lock` 退化为了间隙锁 `(10, 15)`。

SQL语句中条件中使用的是`id`字段（唯一索引），那么根据加锁规则这里不应该用的是优化 2，

而是优化 1，因为优化1中明确指出给唯一索引加锁，从而优化 2的字面意思来理解，它适用于普通索引。不知道是不是我理解的不到位？

2019-01-22

| 作者回复

主要是这里这一行不存在。。

如果能够明确找到一行锁住的话，使用优化1就更准确些

2019-01-23



Justin

0

想咨询一下 普通索引 如果索引中包括的元素都相同 在索引中顺序是怎么排解的呢 是按主键排列的吗 比如(name ,age) 索引 name age都一样 那索引中会按照主键排序吗？

2019-01-22

| 作者回复

会的

2019-01-23



ServerCoder

0

林老师我有个问题想请教一下，描述如下，望给予指点，先谢谢了！

环境：虚拟机，CPU 4核，内存8G，系统CentOS7.4，MySQL版本5.6.40

数据库配置：

bulk_insert_buffer_size = 256M

sql_mode=NO_ENGINESUBSTITUTION,STRICT_TRANS_TABLES

secure_file_priv=""

default-storage-engine=MYISAM

测试场景修改过的参数(以下这些参数得调整对加载效率没有实质的提升)：

myisam_repair_threads=3

myisam_sort_buffer_size=256M

net_buffer_length=1M

myisam_use_mmap=ON

key_buffer_size=256M

测试场景：测试程序多线程，通过客户端API，执行load data infile语句加载数据文件

三个线程，三个文件(每个文件100万条数据、150MB)，三张表(表结构相同，字段类型均为整形，没有定义主键，有一个字段加了非唯一索引)，一一对应进行数据加载，数据库没有使用多核，而是把一个核心的利用率均分给了三个线程。

单个线程加载一个文件大约耗时3秒

单线程加载三个文件到三张表大约耗时9秒

三个线程分别加载三个文件到三张表，则每个线程均耗时大约9秒。从这个效果看，单线程顺序加载和三线程并发加载耗时相同，没有提升效果。

三线程加载过程中查看processlist发现时间主要耗费在了网络读取上。

问题：为啥这种场景下MySQL不利用多核？这种并行加载的情况要如何才能让其利用多核，提

升加载速度

2019-01-22

作者回复

可以用到多核呀，你是怎么得到“时间主要耗费在了网络读取上。”这个结论的？

另外，把这三个文件先拷贝到数据库本地，然后本地执行load看看什么效果？

2019-01-23



慕塔

0

是这样的 假设只有一主一从 1)是集群只有一个sysbench实例，产生的数据流通过中间件，主机分全部写，和30%的读，另外70%的读全部分给从机。2)有两个sysbench，一个读写加压到主机，另一个只有加压到从机。主从复制之间通过binlog。问题在1)的QPS累加与2)QPS累加意义一样吗 1)的一条事务有读写，而2)的情况，主机与1)一样，从机的读事务与主机里的读不一样吧[]

2019-01-22

作者回复

我觉得这两个对比不太公平^_^

1) 的测试可能会出现中间件瓶颈，

- a) 网络环节中间增加了一跳；
 - b) 如果是小查询，可能proxy先打到瓶颈
- 2)的测试结论一般会比1) 好些

但是有这个架构，你肯定是从中间件访问数据库的，所以应该以1的测试结果为准

2019-01-23



Jason_鹏

0

最后一个update的例子，为没有加(0, 5]的间隙呢？我理解应该是先拿c=5去b+树搜索，按照间隙索最右原则，应该会加(0, 5]的间隙，然后c=5不满足大于5条件，根据优化2原则退化成(0, 5)的间隙索，我是这样理解的

2019-01-22

作者回复

根据c>5查到的第一个记录是c=10，因此不会加(0,5]这个next-key lock。

你提醒得对，我应该多说明这句，我加到文稿中啦[]

2019-01-22



长杰

0

老师，之前讲这个例子时，`select * from t where c>=15 and c<=20 order by c desc in share mode;`

最右边加的是(20, 25)的间隙锁，

而这个例子`select * from t where id>10 and id<=15 for update`中，最右边加的是(15,20]的next-k

ey锁，

这两个查询为何最后边一个加的gap锁，一个加的next-key锁，他们都是 \leq 的等值范围查询，区别在哪里？

2019-01-22

| 作者回复

```
select * from t where c>=15 and c<=20 order by c desc in share mode;
```

这个语句是根据 $c=20$ 来查数据的，所以加锁(20,25]的时候，可以使用优化2；

```
select * from t where id>10 and id<=15 for update;
```

这里的id=20，是用“向右遍历”的方式得到的，没有优化，按照“以next-key lock”为加锁单位来执行

2019-01-22



库淘淘

0

对于问题 我理解是这样

session 1:

```
delete from t;
```

```
begin; select * from t for update;
```

session 2:

```
insert into t values(1,1,1);发生等待
```

```
show engine innodb status\G;
```

.....

```
----- TRX HAS BEEN WAITING 5 SEC FOR THIS LOCK TO BE GRANTED:
```

```
RECORD LOCKS space id 75 page no 3 n bits 72 index PRIMARY of table `test`.`t` trx id 75209  
0 lock_mode X insert intention waiting
```

```
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
```

```
0: len 8; hex 73757072656d756d; asc supremum;;
```

其中申请插入意向锁与间隙锁 冲突，supremum这个能否理解为 间隙右边的那个记录

2019-01-21

| 作者回复

发现了

2019-01-22



慕塔

0

大佬 请教一下主从从集群性能测试性能计算问题 如果使用基准测试工具sysbench。数据流有两种

1)

```
sysbench---mycat---mysql主机(读写) TPS QPS1
```

```
| |binlog
```

```
mysql从机(只读)QPS2
```

那性能指标 TPS QPS=QPS1+QPS2

2)

sysbench---mysql主机(读写) TPS QPS1

| binlog

sysbench---mysql从机(只读)TPS QPS2

集群性能指标TPS QPS=QPS1+QPS2

这两种哪种严谨些啊? mycat的损失忽略。

生产中的集群性能怎么算的呢? ? ? (还是学生 谢谢!)

2019-01-21

| 作者回复

TPS就看主库的写入

QPS就看所有从库的读能力加和

不过没看懂你问题中1) 和2) 的区别[]

2019-01-22



HuaMax

0

删除导致锁范围扩大那个例子, `id>10 and id<=15`, 锁范围为什么没有10呢? 不是应该 (5, 10] 吗?

2019-01-21

| 作者回复

不是的, 要找`id>10`的, 并没有命中`id=10`哦, 你可以理解成就是查到了(10,15)这个间隙

2019-01-21



llx

0

回复@往事随风, 顺其自然

前面有解释为什么, 这篇文章有更详细的解释。Gap lock 由右值指定的, 由于 c 不是唯一键, 需要到10, 遍历到10的时候, 就把 5-10 锁了

2019-01-21

| 作者回复

[]

2019-01-21

31 | 误删数据后除了跑路，还能怎么办？

2019-01-23 林晓斌



今天我要和你讨论的是一个沉重的话题：误删数据。

在前面几篇文章中，我们介绍了MySQL的高可用架构。当然，传统的高可用架构是不能预防误删数据的，因为主库的一个**drop table**命令，会通过**binlog**传给所有从库和级联从库，进而导致整个集群的实例都会执行这个命令。

虽然我们之前遇到的大多数的数据被删，都是运维同学或者**DBA**背锅的。但实际上，只要有数据操作权限的同学，都有可能踩到误删数据这条线。

今天我们就来聊聊误删数据前后，我们可以做些什么，减少误删数据的风险，和由误删数据带来的损失。

为了找到解决误删数据的更高效的方法，我们需要先对和MySQL相关的误删数据，做下分类：

1. 使用**delete**语句误删数据行；
2. 使用**drop table**或者**truncate table**语句误删数据表；
3. 使用**drop database**语句误删数据库；
4. 使用**rm**命令误删整个MySQL实例。

误删行

在第24篇文章中，我们提到如果是使用**delete**语句误删了数据行，可以用**Flashback**工具通过闪回把数据恢复回来。

Flashback恢复数据的原理，是修改**binlog**的内容，拿回原库重放。而能够使用这个方案的前提是，需要确保**binlog_format=row** 和 **binlog_row_image=FULL**。

具体恢复数据时，对单个事务做如下处理：

1. 对于**insert**语句，对应的**binlog event**类型是**Write_rows event**，把它改成**Delete_rows event**即可；
2. 同理，对于**delete**语句，也是将**Delete_rows event**改为**Write_rows event**；
3. 而如果是**Update_rows**的话，**binlog**里面记录了数据行修改前和修改后的值，对调这两行的位置即可。

如果误操作不是一个，而是多个，会怎么样呢？比如下面三个事务：

(A)delete ...

(B)insert ...

(C)update ...

现在要把数据库恢复回这三个事务操作之前的状态，用**Flashback**工具解析**binlog**后，写回主库的命令是：

(reverse C)update ...

(reverse B)delete ...

(reverse A)insert ...

也就是说，如果误删数据涉及到了多个事务的话，需要将事务的顺序调过来再执行。

需要说明的是，我不建议你直接在主库上执行这些操作。

恢复数据比较安全的做法，是恢复出一个备份，或者找一个从库作为临时库，在这个临时库上执行这些操作，然后再将确认过的临时库的数据，恢复回主库。

为什么要这么做呢？

这是因为，一个在执行线上逻辑的主库，数据状态的变更往往是有关联的。可能由于发现数据问题的时间晚了一点儿，就导致已经在之前误操作的基础上，业务代码逻辑又继续修改了其他数据。所以，如果这时候单独恢复这几行数据，而又未经确认的话，就可能会出现对数据的二次破坏。

坏。

当然，我们不止要说误删数据的事后处理办法，更重要是要做到事前预防。我有以下两个建议：

1. 把`sql_safe_updates`参数设置为`on`。这样一来，如果我们忘记在`delete`或者`update`语句中写`where`条件，或者`where`条件里面没有包含索引字段的话，这条语句的执行就会报错。
2. 代码上线前，必须经过SQL审计。

你可能会说，设置了`sql_safe_updates=on`，如果我真的要把一个小表的数据全部删掉，应该怎么办呢？

如果你确定这个删除操作没问题的话，可以在`delete`语句中加上`where`条件，比如`where id>=0`。

但是，`delete`全表是很慢的，需要生成回滚日志、写`redo`、写`binlog`。所以，从性能角度考虑，你应该优先考虑使用`truncate table`或者`drop table`命令。

使用`delete`命令删除的数据，你还可以用`Flashback`来恢复。而使用`truncate /drop table`和`drop database`命令删除的数据，就没办法通过`Flashback`来恢复了。为什么呢？

这是因为，即使我们配置了`binlog_format=row`，执行这三个命令时，记录的`binlog`还是`statement`格式。`binlog`里面就只有一个`truncate/drop`语句，这些信息是恢复不出数据的。

那么，如果我们真的是使用这几条命令误删数据了，又该怎么办呢？

误删库/表

这种情况下，要想恢复数据，就需要使用全量备份，加增量日志的方式。这个方案要求线上有定期的全量备份，并且实时备份`binlog`。

在这两个条件都具备的情况下，假如有人中午12点误删了一个库，恢复数据的流程如下：

1. 取最近一次全量备份，假设这个库是一天一备，上次备份是当天0点；
2. 用备份恢复出一个临时库；
3. 从日志备份里面，取出凌晨0点之后的日志；
4. 把这些日志，除了误删除数据的语句外，全部应用到临时库。

这个流程的示意图如下所示：



图1 数据恢复流程-mysqlbinlog方法

关于这个过程，我需要和你说明如下几点：

1. 为了加速数据恢复，如果这个临时库上有多个数据库，你可以在使用mysqlbinlog命令时，加上一个`-database`参数，用来指定误删表所在的库。这样，就避免了在恢复数据时还要应用其他库日志的情况。
2. 在应用日志的时候，需要跳过12点误操作的那个语句的binlog：
 - 如果原实例没有使用GTID模式，只能在应用到包含12点的binlog文件的时候，先用`-stop-position`参数执行到误操作之前的日志，然后再用`-start-position`从误操作之后的日志继续执行；
 - 如果实例使用了GTID模式，就方便多了。假设误操作命令的GTID是`gtid1`，那么只需要执行`set gtid_next=gtid1;begin;commit;`先把这个GTID加到临时实例的GTID集合，之后按顺序执行binlog的时候，就会自动跳过误操作的语句。

不过，即使这样，使用mysqlbinlog方法恢复数据还是不够快，主要原因有两个：

1. 如果是误删表，最好就是只恢复出这张表，也就是只重放这张表的操作，但是mysqlbinlog工具并不能指定只解析一个表的日志；

2. 用mysqlbinlog解析出日志应用，应用日志的过程就只能是单线程。我们在[第26篇文章](#)中介绍的那些并行复制的方法，在这里都用不上。

一种加速的方法是，在用备份恢复出临时实例之后，将这个临时实例设置成线上备库的从库，这样：

1. 在start slave之前，先通过执行

`change replication filter replicate_do_table = (tbl_name)` 命令，就可以让临时库只同步误操作的表；

2. 这样做也可以用上并行复制技术，来加速整个数据恢复过程。

这个过程的示意图如下所示。

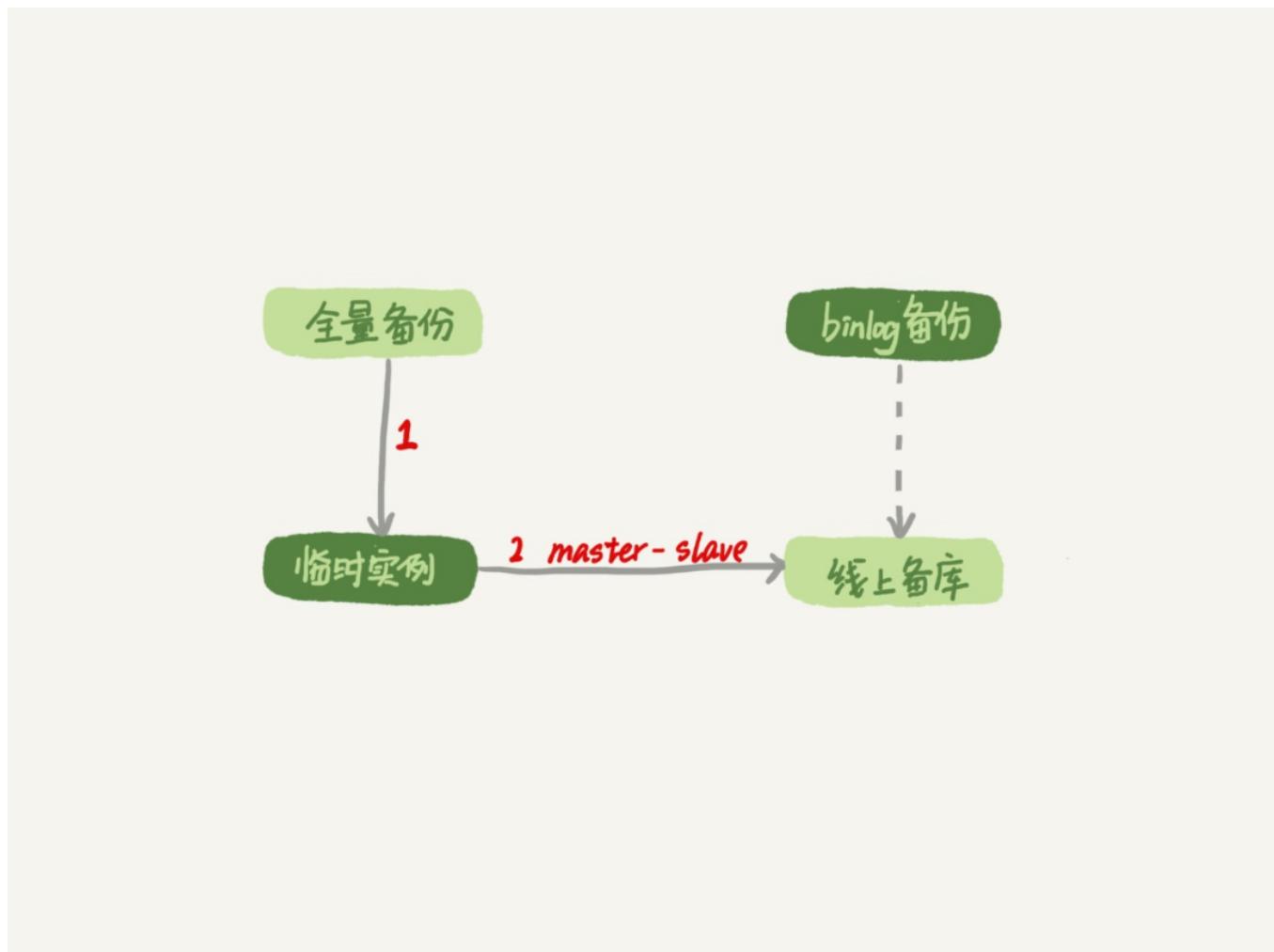


图2 数据恢复流程-master-slave方法

可以看到，图中binlog备份系统到线上备库有一条虚线，是指如果由于时间太久，备库上已经删除了临时实例需要的binlog的话，我们可以从binlog备份系统中找到需要的binlog，再放回备库中。

假设，我们发现当前临时实例需要的binlog是从master.000005开始的，但是在备库上执行show

binlogs 显示的最小的**binlog**文件是**master.000007**, 意味着少了两个**binlog**文件。这时，我们就需要去**binlog**备份系统中找到这两个文件。

把之前删掉的**binlog**放回备库的操作步骤，是这样的：

1. 从备份系统下载**master.000005**和**master.000006**这两个文件，放到备库的日志目录下；
2. 打开日志目录下的**master.index**文件，在文件开头加入两行，内容分别是“**./master.000005**”和“**./master.000006**”；
3. 重启备库，目的是要让备库重新识别这两个日志文件；
4. 现在这个备库上就有了临时库需要的所有**binlog**了，建立主备关系，就可以正常同步了。

不论是把**mysqlbinlog**工具解析出的**binlog**文件应用到临时库，还是把临时库接到备库上，这两个方案的共同点是：误删库或者表后，恢复数据的思路主要就是通过备份，再加上应用**binlog**的方式。

也就是说，这两个方案都要求备份系统定期备份全量日志，而且需要确保**binlog**在被从本地删除之前已经做了备份。

但是，一个系统不可能备份无限的日志，你还需要根据成本和磁盘空间资源，设定一个日志保留的天数。如果你的**DBA**团队告诉你，可以保证把某个实例恢复到半个月内的任意时间点，这就表示备份系统保留的日志时间就至少是半个月。

另外，我建议你不论使用上述哪种方式，都要把这个数据恢复功能做成自动化工具，并且经常拿出来演练。为什么这么说呢？

这里的原因，主要包括两个方面：

1. 虽然“发生这种事，大家都不想的”，但是万一出现了误删事件，能够快速恢复数据，将损失降到最小，也应该不用跑路了。
2. 而如果临时再手忙脚乱地手动操作，最后又误操作了，对业务造成了二次伤害，那就说不过去了。

延迟复制备库

虽然我们可以通过利用并行复制来加速恢复数据的过程，但是这个方案仍然存在“恢复时间不可控”的问题。

如果一个库的备份特别大，或者误操作的时间距离上一个全量备份的时间较长，比如一周一备的实例，在备份之后的第6天发生误操作，那就需要恢复6天的日志，这个恢复时间可能是要按天来计算的。

那么，我们有什么方法可以缩短恢复数据需要的时间呢？

如果有非常核心的业务，不允许太长的恢复时间，我们可以考虑搭建延迟复制的备库。这个功能是MySQL 5.6版本引入的。

一般的主备复制结构存在的问题是，如果主库上有个表被误删了，这个命令很快也会被发给所有从库，进而导致所有从库的数据表也都一起被误删了。

延迟复制的备库是一种特殊的备库，通过 `CHANGE MASTER TO MASTER_DELAY = N` 命令，可以指定这个备库持续保持跟主库有N秒的延迟。

比如你把N设置为3600，这就代表了如果主库上有数据被误删了，并且在1小时内发现了这个误操作命令，这个命令就还没有在这个延迟复制的备库执行。这时候到这个备库上执行`stop slave`，再通过之前介绍的方法，跳过误操作命令，就可以恢复出需要的数据。

这样的话，你就随时可以得到一个，只需要最多再追1小时，就可以恢复出数据的临时实例，也就缩短了整个数据恢复需要的时间。

预防误删库/表的方法

虽然常在河边走，很难不湿鞋，但终究还是可以找到一些方法来避免的。所以这里，我也会给你一些减少误删操作风险的建议。

第一条建议是，账号分离。这样做的目的是，避免写错命令。比如：

- 我们只给业务开发同学DML权限，而不给truncate/drop权限。而如果业务开发人员有DDL需求的话，也可以通过开发管理系统得到支持。
- 即使是DBA团队成员，日常也都规定只使用只读账号，必要的时候才使用有更新权限的账号。

第二条建议是，制定操作规范。这样做的目的，是避免写错要删除的表名。比如：

- 在删除数据表之前，必须先对表做改名操作。然后，观察一段时间，确保对业务无影响以后再删除这张表。
- 改表名的时候，要求给表名加固定的后缀（比如加`_to_be_deleted`），然后删除表的动作必须通过管理系统执行。并且，管理系删除表的时候，只能删除固定后缀的表。

rm删除数据

其实，对于一个有高可用机制的MySQL集群来说，最不怕的就是rm删除数据了。只要不是恶意地把整个集群删除，而只是删掉了其中某一个节点的数据的话，HA系统就会开始工作，选出一个新的主库，从而保证整个集群的正常工作。

这时，你要做的就是在这个节点上把数据恢复回来，再接入整个集群。

当然了，现在不止是**DBA**有自动化系统，**SA**（系统管理员）也有自动化系统，所以也许一个批量下线机器的操作，会让你整个**MySQL**集群的所有节点都全军覆没。

应对这种情况，我的建议只能是说尽量把你的备份跨机房，或者最好是跨城市保存。

小结

今天，我和你讨论了误删数据的几种可能，以及误删后的处理方法。

但，我要强调的是，预防远比处理的意义来得大。

另外，在**MySQL**的集群方案中，会时不时地用到备份来恢复实例，因此定期检查备份的有效性也很有必要。

如果你是业务开发同学，你可以用**show grants**命令查看账户的权限，如果权限过大，可以建议**DBA**同学给你分配权限低一些的账号；你也可以评估业务的重要性，和**DBA**商量备份的周期、是否有必要创建延迟复制的备库等等。

数据和服务的可靠性不止是运维团队的工作，最终是各个环节一起保障的结果。

今天的课后话题是，回忆下你亲身经历过的误删数据事件吧，你用了什么方法来恢复数据呢？你在这个过程中得到的经验又是什么呢？

你可以把你的经历和经验写在留言区，我会在下一篇文章的末尾选取有趣的评论和你一起讨论。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章给你留的问题，是关于空表的间隙的定义。

一个空表就只有一个间隙。比如，在空表上执行：

```
begin;  
select * from t where id>1 for update;
```

这个查询语句加锁的范围就是**next-key lock (-∞, supremum]**。

验证方法的话，你可以使用下面的操作序列。你可以在图4中看到显示的结果。

session A	session B
create table t(id int primary key)engine=innodb; begin; select * from t where id>1;	
	insert into t values(2); (blocked)
show engine innodb status;	

图3 复现空表的next-key lock

```
--TRANSACTION 1300, ACTIVE 13 SEC INSERTING
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 5, OS thread handle 139781560854272, query id 36 localhost 127.0.0.1 root update
insert into t values(2)
----- TRX HAS BEEN WAITING 13 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 24 page no 3 n bits 72 index PRIMARY of table `test`.`t` trx id 1300 lock_mode X insert intention waiting
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum; 
```

图4 show engine innodb status 部分结果

评论区留言点赞板：

@老杨同志 给出了正确的分析和SQL语句验证方法；
 @库淘淘 指出了show engine innodb status验证结论。

赞这些思考和反馈。


极客时间

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL



林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金奖励**。



苍茫

3

有一次，我维护一张表，需要手动修改大量数据的状态，**sql**就很多，然后我保存到**txt**文件中以附件的形式发给部门老大审批，部门老大审批后转发邮件给运维，然后运维这哥们用的是**360**浏览器，他预览的**sql**，然后全部复制到客户端执行，但是问题也在这，**360**浏览器预览的时候由于文本偏长，到了某一条语句只有前半部分的**update**语句，没有后面的条件，然后就悲剧了。全表的状态都变成同一个。然后我就特别莫名其妙，还被老大批了一顿。说我写的脚本有问题。这锅我可不背，我把脚本在本地备份库跑了一遍又一遍就是没有问题。然后我再去运维哥们那，叫他再复制一下脚本就发现问题了。好在执行脚本前进行了表备份。扩展一下，如果你用谷歌浏览器就不会出现这种问题！发现问题后，立马恢复了数据

2019-01-23

作者回复

[]这个是血泪经验

拷贝文本执行，这个操作还可能存在字符集隐患。

这个事情更深一层逻辑，是你做了创造性的事情，非常优秀。

而这个运维同学认为他只是一个“复制粘贴执行的人”，这种思路下是迟早会出问题的。

2019-01-24



linhui0705

1

对生产数据库操作，公司**DBA**提出的编写脚本方法，个人觉得还是值得分享，虽说可能大部分公司也可能有这样的规范。

修改生产的数据，或者添加索引优化，都要先写好四个脚本：备份脚本、执行脚本、验证脚本和回滚脚本。备份脚本是对需要变更的数据备份到一张表中，固定需要操作的数据行，以便误操作或业务要求进行回滚；执行脚本就是对数据变更的脚本，为防**Update**错数据，一般连备份表进行**Update**操作；验证脚本是验证数据变更或影响行数是否达到预期要求效果；回滚脚本就是将数据回滚到修改前的状态。

虽说分四步骤写脚本可能会比较繁琐，但是这能够很大程度避免数据误操作。

2019-01-23

作者回复

[]非常好的经验

如果能够切实执行，即使有出问题，也是可以很快恢复的

把这些脚本当做开发代码来维护，是一个很好的实践

2019-01-23



某、人

9

总结下今天的知识点：

我觉得**DBA**的最核心的工作就是保证数据的完整性

今天老师也讲到了先要做好预防，预防的话大概是通过这几个点：

1. 权限控制与分配(数据库和服务器权限)

- 2.制作操作规范
 - 3.定期给开发进行培训
 - 4.搭建延迟备库
 - 5.做好sql审计,只要是对线上数据有更改操作的语句(DML和DDL)都需要进行审核
 - 6.做好备份。备份的话又分为两个点。
 - (1)如果数据量比较大,用物理备份xtrabackup。定期对数据库进行全量备份,也可以做增量备份。
 - (2)如果数据量较少,用mysqldump或者mysqldumper。再利用binlog来恢复或者搭建主从的方式来恢复数据。
- 定期备份binlog文件也是很有必要的
还需要定期检查备份文件是否可用,如果真的发生了误操作,需要恢复数据的时候,发生备份文件不可用,那就更悲剧了

如果发生了数据删除的操作,又可以从以下几个点来恢复:

1.DML误操作语句造成数据不完整或者丢失。可以通过flashback,不过我们目前用的是美团的myflash,也是一个不错的工具,本质都差不多.都是先解析binlog event,然后在进行反转。把delete反转为insert,insert反转为delete,update前后image对调。所以必须设置binlog_format=row 和 binlog_row_image=full.

切记恢复数据的时候,应该先恢复到临时的实例,然后在恢复回主库上。

2.DDL语句误操作(truncate和drop),由于DDL语句不管binlog_format是row还是statement.在binlog里都只记录语句,不记录image所以恢复起来相对要麻烦得多。只能通过全量备份+应用binlog的方式来恢复数据。一旦数据量比较大,那么恢复时间就特别长,

对业务是个考验。所以就涉及到老师在第二讲提到的问题了,全量备份的周期怎么去选择

2019-01-23

作者回复

0

2019-01-23



亮

2

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `city` varchar(16) NOT NULL,
  `name` varchar(16) NOT NULL,
  `age` int(11) NOT NULL,
  `addr` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `city` (`city`)
) ENGINE=InnoDB;
```

老师请教您16章的问题,您提到“city、name、age这三个字段的定义总长度是36”,这个是怎么算出来的呢,varchar(16)是可以保存16个字符,占用了49个字节(utf8),所以我没想明白36是怎么来的。

第二个问题是max_length_for_sort_data参数系统默认是1024,是1024个字节的意思吗?

2019-01-23

|| 作者回复

1. `age(11)`其实是4个字节哈

2. 对，单位是字节

谢谢老师，不过还是没明白，`age`是4个字节，`city`和`name`分别是49个字节， $49+49+4=102$ 字节，36是怎么来的呢？再次感谢

2019-01-23

|| 作者回复

哦 抱歉哈，我这边验证的时候默认用的`latin1`，是 $16+16+4$

2019-01-23



技术人成长

1

我只想说，作者功力过于深厚了！

2019-01-25



Cranliu

1

个人觉得，预防同样很重要，一般的dml操作，我是先`ctas`要操作的数据，`drop/truncate`的时候先逻辑备份。

2019-01-23

|| 作者回复

对的，备份的意识很重要。

不过“`drop/truncate` 的时候先逻辑备份”这么做的不多^_^

主要的原因是逻辑备份可能会对系统有额外消耗。（全表扫描）

2019-01-23



511

1

早~

2019-01-23



Long

0

又到了讲故事(事故)的时候了，历史上遇到过很多次事故。全表误删除，误更新不下于8次，有MySQL的DB也有memory DB.有一次同事比较搞笑的是，有一次一张重要的权限控制表更新，由于用的是workbench界面工具当时写了`where`条件，但是在选中执行行的时候`where`条件在第二行，没选中，还在执行前的时候手动把session级的`sql_safe_updates=0`了，也没有点开那个`autocommit`取消的按钮。然后一执行，全表更新了，导致全网只有一个用户可以正常登录。还有其他的误操作，总结历史遇到过的这类问题基本就是几类

1. 登错环境，以为是测试环境，一顿操作猛如虎，一看环境是生产，回头一看，表已经drop了.....
2. sql写的有问题，逻辑错误，或者条件缺失，常见的如不带where; or关键字的逻辑没有用括号括好
3. 还有一些奇葩的，比如where 字段1=字段2写成了字段1+字段2，逻辑等于判断变成了是否为1的判断了，大概率全表更新了。

错误解决大部分都是用备份恢复或者根据错误的逻辑来逻辑恢复。

还有一个，最近在尝试的，就是ibd文件中有坏页，只要一读到那个坏页，就会crash，报错spaceid page no should be多少多少，尝试了copy frm, ibd, ibdata, iblogfile这些表结构，数据文件，数据字典，undo redo 日志，也尝试用了undrop的工具也解析不出来。这个表比较特殊，是一个特殊库，没备份，表没有索引没法通过走索引跳过那个坏页的那些行，现在的状态是，只能用mysqldump恢复一部分数据。我想通过16进制，自己慢慢找到那个脏写的数据，然后修改一下文件.....

老师有什么比较好的建议吗？或者后面会说到ibd文件的物理结构之类的吗？感谢

2019-01-28

作者回复

感谢你的分享，都是血泪教训。。

我看有几个是用的可视化工具导致的，后面还是尽量用MySQL客户端敲命令吧！

ibd文件坏页我之前有回答过其他同学的，看下这个

https://weibo.com/1933424965/H3qlu0JYo?from=page_1005051933424965_profile&wvr=6&mod=weibotime

2019-01-28



PengfeiWang

0

老师，您好，有个问题请教一下：

关于MySQL备份有效性的验证，你有什么好的方法可以推荐吗？目前只能通过不定期的备份恢复来验证。

2019-01-25

作者回复

大家都是这么做的！

2019-01-25



AI杜嘉嘉

0

老师，接着上面问题。是删表，线上rename，有什么风险吗？需要注意什么？rename是不是ddl操作

2019-01-25

作者回复

是，不过rename的执行速度很快

2019-01-25



还一棵树

0

我遇到过一个线上误truncate表的，最终选择的处理过程如下：

- 1、创建一个同版本的空mysql实例，建一个名字+结构一模一样的表
- 2、**discard**这个表的tablespace
- 3、从之前的备份集中 innobackupex --apply-log 并记录binlog位置（用innobackupex备份的）。还原后找到误操作表的.ibd文件，copy到新实例对应的位置
- 4、在之前创建的mysql实例上import tablespace
- 5、利用mysqlbinlog 处理增量数据
- 6、最后导出 再导入

2019-01-24

| 作者回复

[]

这基本上是最快的恢复步骤了

2019-01-24



苍茫

0

有一次我在查询数据倒数报表给业务方，那个脚本是我写的，关联了很多表，还跨了库，一个主表有一万多条纪录，关联另一张操作记录表好像是10万条数据。因为要统计多步操作步骤，所以每一步的操作记录我就得按照不同的条件关联产生临时表（关联中有group 还有max()聚合函数，这个是需求导致的），一开始写好的查询很快有了结果。那天11点多的时候，我执行那个脚本，发现很慢没有反应，然后我就把连接关了，重复几次操作，然后生产库就被我搞挂了。后面运维的同学操作了一波才恢复过来。这次也是运维同学背的锅。后面，还把我的操作给贴出来了，做通报批评。我想问下为啥会出现这种情况呢？

后续我的组长对我写的sql进行了优化，主要是把联表操作需要的信息放在子查询中，然后再操作记录表中加了索引，有了备份库，每次执行脚本导出数据都是在备份库中导出，就再也没有发生这个问题了。

2019-01-24

| 作者回复

后面有一篇专门说这个，敬请期待哈

2019-01-24



xishuai

0

老师，麻烦问一下，5.7.21上innodb的表两列（有中文有英文）建的全文索引，最小分词1，按中文可以查询，按英文有些查询不出来，您知道原因吗？

2019-01-24

| 作者回复

全文索引有stop words的，你看看是不是落在stop words里了

2019-01-24



catalina

0

老师，我们现在需要将一个库下面的所有表的数据同步到另外一个库，每个表有几百万数据吧，大约十多张表。有什么好的方法吗？

2019-01-24

作者回复

原库的这几个表还会继续更新吗？如果会继续更新，就用搭主备的方法；

如果没更新了，后面有一个文章专门讲这个问题哈

2019-01-24



hua168

0

大神，有亲戚小公司搞**DBA**一年，我想问一下：

1.**DBA**一般发展方向是怎样的呀？运维和开发我了解，**DBA**没接触过，无法给建议，一般的升级过程是怎样的？

2.以后发展方向是怎样？现在都是开源、大数据时代时代，阿里又搞“去IOE”，一般**oracle DBA**发展前景不好吧？

能帮指一个大概的方向吗？谢谢~~

2019-01-24



aliang

0

老师好。这是第6讲评论区Tony Du的评论：

session A: begin; select * from t limit 1; 最先启动**sessionA**

session B: begin; select * from t limit 1; 紧接着启动**sessionB**

session C: alter table t add f int; 然后再是启动**sessionC**

session D: begin; select * from t limit 1; 最后是启动**sessionD**

他说**session C**会被**A**和**B**阻塞，**D**会被**C**阻塞；当**A**和**B**提交后，**D**是可以继续执行得到查询结果的，但是**C**仍然被阻塞，只有**D**提交后**C**才能执行成功。我自己在**5.6**和**5.7**按他的步骤做了试验，结果和他一样。

然后我再做了一次试验，这次把**D**的**begin;**去掉，变成了：

session A: begin; select * from t limit 1; 最先启动**sessionA**

session B: begin; select * from t limit 1; 紧接着启动**sessionB**

session C: alter table t add f int; 然后再是启动**sessionC**

session D: select * from t limit 1; 最后是启动**sessionD**

结果是当**A**和**B**提交后，**D**和**C**都能执行成功了（和老师的结果一样）。我的问题是：为什么第一次**session D**显式开启事务，和第二次不显式开启的结果不一样呢？

2019-01-24

作者回复

可否贴一下你的**show variables** 的结果，我这边验证（不论**D**有没有加**begin**）的效果都是你说的第二次的情况哦

2019-02-04



太福

0

因为时间原因，前面的课程没跟上，在这里请教个最近线上**mysql**遇到的问题：

一个从库**mysql5.6**版本，正常情况下只有3到5个并发**sql**在查询，每分钟用下面的**sql**查看一次检测到的

```
SELECT t1.* FROM information_schema.`PROCESSLIST` t1 WHERE info is not null ORDER BY time desc;
```

前一次检测还是几个sql在查询，下1分钟查到2000多个sql在跑，很多堆积了几十秒的sql，状态“Sending data” “Creating sort index”

，而这些sql在正常情况下是不到1秒就查到结果了的，且cpu使用率与io很低，看起来mysql僵死的了。

有两个问题：1) 问题突然出现

2) 大量sql在跑，而cpu与磁盘io反而比正常下降；这是从库，写只有主从同步，其它都是读查询。

配置：64G内存，bp分配40G，io使用不高

，业务量没大变动，也没新版本发布，大体排除业务并发加大导致的。

2019-01-24

| 作者回复

这个不算突然出现吧，你两次检测之间是1分钟是吧？

你这样说我看不太明白，可否给一个当时的show processlist的截图；

你还能执行show processlist，就不能算“僵死”；

io的状态如果有保存，也贴一下当时的iostat的结果；

可以发个微博附图，然后地址发到评论区哈

2019-02-01



风二中

0

老师，您好。如何设置binlog 的备份时间呢，感觉RPO 时间总是不能为零，如果是informix 可以只丢一个逻辑日志。对于需要保证mysql恢复 RPO 时间为零，有什么建议吗？备库延迟1小时，加每小时备份一次binlog 。

2019-01-23

| 作者回复

嗯 对于核心业务，使用延迟复制的备份

RPO时间为0？有这么凶残的业务需求吗。。我能想到的就是多套延迟备份的库。

比如开3个，一个是10分钟，一个20分钟，一个30分钟（主要考虑成本）

RPO这么敏感的，应该有对应敏感的监控，误操作要是30分钟还不能发现，可以挑战一下，这个业务是不是值得这么高的指标^_^

2019-01-23



700

0

老师，请教。假如我有数据库的物理备份和逻辑备份（mydumper），因为 mydumper 导出的数据是按表名分开存放的，那么表误删数据的时候优先考虑逻辑备份（误删数据表的备份集）+binlog 恢复比物理备份恢复会快点？基于此，我总感觉物理备份只是在要恢复整个实例时才会优先考虑，而恢复整个实例的场景又是比较少的，毕竟一般大家的线上架构至少都是主从模式。所以逻辑备份被物理备份更实用。这种想法算是说得通吗？

2019-01-23

| 作者回复

其实是要看表的大小

如果是一个大表，逻辑恢复还是比较慢的，毕竟用物理备份来恢复出实例，相对会快些。

当然如果你已经有了一个成熟系统用逻辑恢复来实现，也不用改它，主要关注一下是否满足**SLA**就可以了^_^

facebook就是主要用逻辑备份的

2019-01-23



高强

0

老师你好，问个带子查询的**delete/update/insert**问题，**Delete from A where name in(**

Select name from B where time<'2019-01-23 11:11:12'

) 这条语句删除**A**表记录之前是不是也会把表 **B**满足条件的记录也会给锁住呢？

我试验了一下会锁住**B**表记录的，有没有其他办法不让锁**B**表呢？

2019-01-23

| 作者回复

改成**RC**隔离级别试试

2019-01-23

32 | 为什么还有kill不掉的语句？

2019-01-25 林晓斌



在MySQL中有两个kill命令：一个是**kill query +线程id**，表示终止这个线程中正在执行的语句；一个是**kill connection +线程id**，这里connection可缺省，表示断开这个线程的连接，当然如果这个线程有语句正在执行，也是要先停止正在执行的语句的。

不知道你在使用MySQL的时候，有没有遇到过这样的现象：使用了**kill**命令，却没能断开这个连接。再执行**show processlist**命令，看到这条语句的**Command**列显示的是**Killed**。

你一定会奇怪，显示为**Killed**是什么意思，不是应该直接在**show processlist**的结果里看不到这个线程了吗？

今天，我们就来讨论一下这个问题。

其实大多数情况下，**kill query/connection**命令是有效的。比如，执行一个查询的过程中，发现执行时间太久，要放弃继续查询，这时我们就可以用**kill query**命令，终止这条查询语句。

还有一种情况是，语句处于锁等待的时候，直接使用**kill**命令也是有效的。我们一起来看下这个例子：

session A	session B	session C
begin; update t set c=c+1 where id=1;		
	update t set c=c+1 where id=1; (blocked)	
	ERROR 1317 (70100): Query execution was interrupted	kill query thread_id_B;

图1 kill query 成功的例子

可以看到，**session C** 执行**kill query**以后，**session B**几乎同时就提示了语句被中断。这，就是我们预期的结果。

收到**kill**以后，线程做什么？

但是，这里你要停下来想一下：**session B**是直接终止掉线程，什么都不管就直接退出吗？显然，这是不行的。

我在[第6篇文章](#)中讲过，当对一个表做增删改查操作时，会在表上加**MDL**读锁。所以，**session B**虽然处于**blocked**状态，但还是拿着一个**MDL**读锁的。如果线程被**kill**的时候，就直接终止，那之后这个**MDL**读锁就没机会被释放了。

这样看来，**kill**并不是马上停止的意思，而是告诉执行线程说，这条语句已经不需要继续执行了，可以开始“执行停止的逻辑了”。

其实，这跟**Linux**的**kill**命令类似，**kill -N pid**并不是让进程直接停止，而是给进程发一个信号，然后进程处理这个信号，进入终止逻辑。只是对于**MySQL**的**kill**命令来说，不需要传信号量参数，就只有“停止”这个命令。

实现上，当用户执行**kill query thread_id_B**时，**MySQL**里处理**kill**命令的线程做了两件事：

1. 把**session B**的运行状态改成**THD::KILL_QUERY**(将变量**killed**赋值为**THD::KILL_QUERY**);
2. 给**session B**的执行线程发一个信号。

为什么要发信号呢？

因为像图1的我们例子里面，**session B**处于锁等待状态，如果只是把**session B**的线程状态设置**THD::KILL_QUERY**，线程B并不知道这个状态变化，还是会继续等待。发一个信号的目的，就是让**session B**退出等待，来处理这个**THD::KILL_QUERY**状态。

上面的分析中，隐含了这么三层意思：

1. 一个语句执行过程中有多处“埋点”，在这些“埋点”的地方判断线程状态是THD::KILL_QUERY，才开始进入语句终止逻辑；
2. 如果处于等待状态，必须是一个可以被唤醒的等待，否则根本不会执行到“埋点”处；
3. 语句从开始进入终止逻辑，到终止逻辑完全完成，是有一个过程的。

到这里你就知道了，原来不是“说停就停的”。

接下来，我们再看一个kill不掉的例子，也就是我们在前面[第29篇文章](#)中提到的innodb_thread_concurrency不够用的例子。

首先，执行set global innodb_thread_concurrency=2，将InnoDB的并发线程上限数设置为2；然后，执行下面的序列：

session A	session B	session C	session D	session E
select sleep(100) from t;	select sleep(100) from t;			
		select *from t; (blocked)		
			kill query C;	
		ERROR 2013 (HY000): Lost connection to MySQL server during query		kill C;

图2 kill query 无效的例子

可以看到：

1. session C执行的时候被堵住了；
2. 但是session D执行的kill query C命令却没什么效果，
3. 直到session E执行了kill connection命令，才断开了session C的连接，提示“Lost connection to MySQL server during query”，
4. 但是这时候，如果在session E中执行show processlist，你就能看到下面这个图。

mysql> show processlist;							
Id	User	Host	db	Command	Time	State	Info
4	root	localhost:50934	test	Query	30	User sleep	select sleep(100) from t
5	root	localhost:50956	test	Query	26	User sleep	select sleep(100) from t
12	root	localhost:53288	test	Killed	24	Sending data	select * from t
13	root	localhost:53316	test	Query	0	starting	show processlist

图3 kill connection之后的效果

这时候，`id=12`这个线程的`Command`列显示的是`Killed`。也就是说，客户端虽然断开了连接，但实际上服务端上这条语句还在执行过程中。

为什么在执行`kill query`命令时，这条语句不像第一个例子的`update`语句一样退出呢？

在实现上，等行锁时，使用的是`pthread_cond_timedwait`函数，这个等待状态可以被唤醒。但是，在这个例子里，12号线程的等待逻辑是这样的：每10毫秒判断一下是否可以进入InnoDB执行，如果不行，就调用`nanosleep`函数进入`sleep`状态。

也就是说，虽然12号线程的状态已经被设置成了`KILL_QUERY`，但是在这个等待进入InnoDB的循环过程中，并没有去判断线程的状态，因此根本不会进入终止逻辑阶段。

而当session E执行`kill connection`命令时，是这么做的，

1. 把12号线程状态设置为`KILL_CONNECTION`；
2. 关掉12号线程的网络连接。因为有这个操作，所以你会看到，这时候session C收到了断开连接的提示。

那为什么执行`show processlist`的时候，会看到`Command`列显示为`killed`呢？其实，这就是因为在执行`show processlist`的时候，有一个特别的逻辑：

如果一个线程的状态是`KILL_CONNECTION`，就把`Command`列显示成`Killed`。

所以其实，即使是客户端退出了，这个线程的状态仍然是在等待中。那这个线程什么时候会退出呢？

答案是，只有等到满足进入InnoDB的条件后，session C的查询语句继续执行，然后才有可能判断到线程状态已经变成了`KILL_QUERY`或者`KILL_CONNECTION`，再进入终止逻辑阶段。

到这里，我们来小结一下。

这个例子是`kill`无效的第一类情况，即：线程没有执行到判断线程状态的逻辑。跟这种情况相同的，还有由于IO压力过大，读写IO的函数一直无法返回，导致不能及时判断线程的状态。

另一类情况是，终止逻辑耗时较长。这时候，从`show processlist`结果上看也是`Command=Killed`，需要等到终止逻辑完成，语句才算真正完成。这类情况，比较常见的场景有以下几种：

1. 超大事务执行期间被`kill`。这时候，回滚操作需要对事务执行期间生成的所有新数据版本做回收操作，耗时很长。
2. 大查询回滚。如果查询过程中生成了比较大的临时文件，加上此时文件系统压力大，删除临时文件可能需要等待IO资源，导致耗时较长。
3. DDL命令执行到最后阶段，如果被`kill`，需要删除中间过程的临时文件，也可能受IO资源影响耗时较久。

之前有人问过我，如果直接在客户端通过`Ctrl+C`命令，是不是就可以直接终止线程呢？

答案是，不可以。

这里有一个误解，其实在客户端的操作只能操作到客户端的线程，客户端和服务端只能通过网络交互，是不可能直接操作服务端线程的。

而由于MySQL是停等协议，所以这个线程执行的语句还没有返回的时候，再往这个连接里面继续发命令也是没有用的。实际上，执行`Ctrl+C`的时候，是MySQL客户端另外启动一个连接，然后发送一个`kill query`命令。

所以，你可别以为在客户端执行完`Ctrl+C`就万事大吉了。因为，要`kill`掉一个线程，还涉及到后端的很多操作。

另外两个关于客户端的误解

在实际使用中，我也经常会碰到一些同学对客户端的使用有误解。接下来，我们就来看看两个最常见的误解。

第一个误解是：如果库里面的表特别多，连接就会很慢。

有些线上的库，会包含很多表（我见过最多的一个库里有6万个表）。这时候，你就会发现，每次用客户端连接都会卡在下面这个界面上。

```
mysql -h127.0.0.1 -uu1 -pp1 db1
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
```

图4 连接等待

而如果`db1`这个库里表很少的话，连接起来就会很快，可以很快进入输入命令的状态。因此，有

同学会认为是表的数目影响了连接性能。

从[第一篇文章](#)你就知道，每个客户端在和服务端建立连接的时候，需要做的事情就是TCP握手、用户校验、获取权限。但这几个操作，显然跟库里面表的个数无关。

但实际上，正如图中的文字提示所说的，当使用默认参数连接的时候，MySQL客户端会提供一个本地库名和表名补全的功能。为了实现这个功能，客户端在连接成功后，需要多做一些操作：

1. 执行**show databases;**
2. 切到db1库，执行**show tables;**
3. 把这两个命令的结果用于构建一个本地的哈希表。

在这些操作中，最花时间的就是第三步在本地构建哈希表的操作。所以，当一个库中的表个数非常多的时候，这一步就会花比较长的时间。

也就是说，我们感知到的连接过程慢，其实并不是连接慢，也不是服务端慢，而是客户端慢。

图中的提示也说了，如果在连接命令中加上-A，就可以关掉这个自动补全的功能，然后客户端就可以快速返回了。

这里自动补全的效果就是，你在输入库名或者表名的时候，输入前缀，可以使用Tab键自动补全表名或者显示提示。

实际使用中，如果你自动补全功能用得并不多，我建议你每次使用的时候都默认加-A。

其实提示里面没有说，除了加-A以外，加-quick(或者简写为-q)参数，也可以跳过这个阶段。但是，这个-quick是一个更容易引起误会的参数，也是关于客户端常见的一个误解。

你看到这个参数，是不是觉得这应该是一个让服务端加速的参数？但实际上恰恰相反，设置了这个参数可能会降低服务端的性能。为什么这么说呢？

MySQL客户端发送请求后，接收服务端返回结果的方式有两种：

1. 一种是本地缓存，也就是在本地开一片内存，先把结果存起来。如果你用API开发，对应的就是**mysql_store_result**方法。
2. 另一种是不缓存，读一个处理一个。如果你用API开发，对应的就是**mysql_use_result**方法。

MySQL客户端默认采用第一种方式，而如果加上-quick参数，就会使用第二种不缓存的方式。

采用不缓存的方式时，如果本地处理得慢，就会导致服务端发送结果被阻塞，因此会让服务端变慢。关于服务端的具体行为，我会在下一篇文章再和你展开说明。

那你会说，既然这样，为什么要给这个参数取名叫作**quick**呢？这是因为使用这个参数可以达到以下三点效果：

- 第一点，就是前面提到的，跳过表名自动补全功能。
- 第二点，`mysql_store_result`需要申请本地内存来缓存查询结果，如果查询结果太大，会耗费较多的本地内存，可能会影响客户端本地机器的性能；
- 第三点，是不会把执行命令记录到本地的命令历史文件。

所以你看到了，`-quick`参数的意思，是让客户端变得更快。

小结

在今天这篇文章中，我首先和你介绍了MySQL中，有些语句和连接“kill不掉”的情况。

这些“kill不掉”的情况，其实是因为发送**kill**命令的客户端，并没有强行停止目标线程的执行，而只是设置了个状态，并唤醒对应的线程。而被**kill**的线程，需要执行到判断状态的“埋点”，才会开始进入终止逻辑阶段。并且，终止逻辑本身也是需要耗费时间的。

所以，如果你发现一个线程处于**Killed**状态，你可以做的事情就是，通过影响系统环境，让这个**Killed**状态尽快结束。

比如，如果是第一个例子里**InnoDB**并发度的问题，你就可以临时调大
`innodb_thread_concurrency`的值，或者停掉别的线程，让出位子给这个线程执行。

而如果是回滚逻辑由于受到**IO**资源限制执行得比较慢，就通过减少系统压力让它加速。

做完这些操作后，其实你已经没有办法再对它做什么了，只能等待流程自己完成。

最后，我给你留下一个思考题吧。

如果你碰到一个被**killed**的事务一直处于回滚状态，你认为是应该直接把MySQL进程强行重启，还是应该让它自己执行完成呢？为什么呢？

你可以把你的结论和分析写在留言区，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章末尾，给你留下的问题是，希望你分享一下误删数据的处理经验。

@苍茫 同学提到了一个例子，我觉得值得跟大家分享一下。运维的同学直接拷贝文本去执

行，SQL语句截断，导致数据库执行出错。

从浏览器拷贝文本执行，是一个非常不规范的操作。除了这个例子里面说的SQL语句截断问题，还可能存在乱码问题。

一般这种操作，如果脚本的开发和执行不是同一个人，需要开发同学把脚本放到git上，然后把git地址，以及文件的md5发给运维同学。

这样就要求运维同学在执行命令之前，确认要执行的文件的md5，跟之前开发同学提供的md5相同才能继续执行。

另外，我要特别点赞一下@苍茫同学复现问题的思路和追查问题的态度。

@linhui0705同学提到的“四个脚本”的方法，我非常推崇。这四个脚本分别是：备份脚本、执行脚本、验证脚本和回滚脚本。如果能够坚持做到，即使出现问题，也是可以很快恢复的，一定能降低出现故障的概率。

不过，这个方案最大的敌人是这样的思想：这是个小操作，不需要这么严格。

@Knight²⁰¹给了一个保护文件的方法，我之前没有用过这种方法，不过这确实是一个不错的思路。

为了数据安全和服务稳定，多做点预防方案的设计讨论，总好过故障处理和事后复盘。方案设计讨论会和故障复盘会，这两种会议的会议室气氛完全不一样。经历过的同学一定懂的。

The image shows the front cover of a book titled "MySQL 实战 45 讲" (MySQL Practical 45 Lectures) by Lin Xiaobin (林晓斌). The cover features a portrait of the author, a man with glasses and a black shirt, standing with his arms crossed. The title is prominently displayed in large, bold, dark font. Below the title, a subtitle reads "从原理到实战，丁奇带你搞懂 MySQL" (From principle to practical, Dingqi leads you to understand MySQL). At the bottom left, the author's name "林晓斌" is listed along with the text "网名丁奇" and "前阿里资深技术专家". A promotional message at the bottom right encourages users to upgrade to a new version and share it with friends for free reading.

极客时间

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌 网名丁奇 前阿里资深技术专家

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。



Leon

2

kill connection本质上只是把客户端的sql连接断开，后面的执行流程还是要走**kill query**的，是这样理解吧

2019-01-30

作者回复

这个理解非常到位

额外的一个不同就是**show processlist**的时候，**kill connection**会显示“killed”

这两句加起来可以用来替换我们文中的描述

2019-01-30



Mr.sylar

2

老师，我想问下这些原理的“渔”的方法除了看源码，还有别的建议吗

2019-01-25

作者回复

不同的知识点不太一样哈，

有些可以看文档；

有些可以自己验证；

还有就是看其他人文章，加验证；（就是我们这个专栏的方法^_^）

2019-01-25



夹心面包

2

对于结尾的问题，我觉得肯定是等待，即便是**mysql**重启，也是需要对未提交的事务进行回滚操作的，保证数据库的一致性

2019-01-25



Ryoma

1

想得简单点：既然事务处于回滚状态了，重启**MySQL**这部分事务还是需要回滚。私以为让它执行完成比较好。

2019-01-25



斜面镜子 Bill

0

“采用不缓存的方式时，如果本地处理得慢，就会导致服务端发送结果被阻塞，因此会让服务端变慢”这个怎么理解？

2019-01-28

作者回复

堵住了不就变慢了

2019-01-28



700

0



老师，您好。客户端版本如下：

mysql Ver 14.14 Distrib 5.7.24, for linux-glibc2.12 (x86_64) using EditLine wrapper

老师，再请教另一个问题。并非所有的 **DDL** 操作都可以通过主从切换来实现吧？不适用的场景有哪些呢？

2019-01-27

作者回复

对，其实只有 改索引、加最后一列、删最后一列

其他的大多数不行，比如删除中间一列这种

2019-01-28



千年孤独

0

可能不是本章讨论的问题，我想请问老师“MySQL使用自增ID和UUID作为主键的优劣”，基于什么样的业务场景用哪种好？

2019-01-27

作者回复

后面会有文章会提到这个问题哈：）

2019-01-27



Geek_a67865

0

老师好，我猜发条橙子的问题 因为很多日志监控会统计**error**日志，这样并不很优雅，觉得他是想有什么办法规避这种并发引起的问题，

2019-01-26

作者回复

嗯嗯 不过我也确实没有想到更好的方法

毕竟两个线程要同时发起一个**insert**操作，这个服务端也拦不住呀！

2019-01-26



路过

0

老师，**kill**语法是：

KILL [CONNECTION | QUERY] processlist_id

processlist_id是**conn_id**, 不是**thd_id**. 通过对**sys.processlist**表中的信息就可以知道了。

通过查询官方文档也说明了：

thd_id: The thread ID.

conn_id: The connection ID.

所以，这篇文章开头的：

在 MySQL 中有两个 **kill** 命令：一个是 **kill query + 线程 id**

感觉有点不对。请老师指正。谢谢！

2019-01-26

作者回复

这两个是一样的吧？

都是对应**show processlist**这个命令结果里的第一列

2019-01-26



HuaMax

0

课后题。我认为需要看当时的业务场景。重启会导致其他的连接也断开，返回给其他业务连接丢失的错误。如果有很多事务在等待该事务的锁，则应该重启，让其他事务快速重试获取锁。另外如果是RR的事务隔离级别，长事务会因为数据可见性的问题，对于多版本的数据需要找到正确的版本，对读性能是不是也会有影响，这时候重启也更好。个人理解，请老师指正。

2019-01-26

作者回复

有考虑到对其他线程的影响，这个

其实这种时候往往是要先考虑切换（当然重启也是切换的）

如果只看恢复时间的话，等待会更快

2019-01-26



Geek_a67865

0

也遇到@发条橙子一样的问题，例如队列两个消息同时查询库存，发现都不存在，然后就都执行插入语句，一条成功，一条报唯一索引异常，这样程序日志会一直显示一个唯一索引报错，然后重试执行更新，我暂时是强制查主库

2019-01-26

作者回复

“我暂时是强制查主库”从这就看你是因为读是读的备库，才出现这个问题的是吧。

发条橙子的问题是，他都是操作主库。

其实如果索引有唯一键，就直接上insert。

然后碰到违反唯一键约束就报错，这个应该就是唯一键约束正常的用法吧

2019-01-26



gaohueric

0

老师您好，一个表中1个主键，2个唯一索引，1个普通索引4个普通字段，当插入一条全部字段不为空的数据时，此时假设有4个索引文件，分别对应主键唯一性索引，普通索引，假设内存中没有这个数据页，那么server是直接调用innodb的接口，然后依次校验（读取磁盘数据，验证唯一性）主键，唯一性索引，然后确认无误A时刻之后，吧主键和唯一性索引的写入内存，再把普通索引写入change buffer？那普通数据呢，是不是跟着主键一块写入内存了？

2019-01-26

作者回复

1. 是的，如果普通索引上的数据页这时候没有在内存中，就会使用change buffer

2. “那普通数据呢，是不是跟着主键一块写入内存了？”你说的是无索引的字段是吧，这些数据就在主键索引上，其实改的就是主键索引。

2019-01-26



700

0

老师，您好。我继续接着我上条留言。

关于2），因为是测试机，我是直接 tail -Of 观察 general log 输出的。确实没看到 KILL QUERY 等字眼。数据库版本是 MySQL 5.7.24。

关于4），文中您不是这样写的吗？

2.但是 session D 执行的 kill query C 命令却没什么效果，

3.直到 session E 执行了 kill connection 命令，才断开了 session C 的连接，提示“Lost connection to MySQL server during query”，

感谢您的解答。

2019-01-26

作者回复

1. 你的客户端版本是什么 mysql --version 看看

3. 嗯，是的，连接会断开，但是这个语句在server端还是会继续执行（如果kill query 无效的话）

2019-01-26



700

0

老师，请教。

1) 文中开头说“当然如果这个线程有语句正在执行，也是要先停止正在执行的语句的”。我个人在平时使用中就是按默认的执行，不管这个线程有无正在执行语句。不知这样会有什么潜在问题？

2) 文中说“实际上，执行 Ctrl+C 的时候，是 MySQL 客户端另外启动一个连接，然后发送一个 kill query 命令”。这个怎么解释呢？

我开启 general log 的时候执行 Ctrl+C 或 Ctrl+D 并没有看到有另外启动一个连接，也没有看到 kill query 命令。general log 中仅看到对应线程 id 和 Quit。

3) MySQL 为什么要同时存在 kill query 和 kill connection，既然 kill query 有无效的场景，干嘛不直接存在一个 kill connection 命令就好了？那它俩分别对应的适用场景是什么，什么时候考虑 kill query，什么时候考虑 kill connection？我个人觉得连接如果直接被 kill 掉大不了再重连一次好了。也没啥损失。

4) 小小一个总结，不知对否？

kill query - 会出现无法 kill 掉的情况，只能再次执行 kill connection。

kill connection - 会出现 Command 列显示成 Killed 的情况。

2019-01-25

作者回复

1. 一般你执行kill就是要停止正在执行的语句，所以问题不大

2. 不应该呀， KILL QUERY 是大写哦，你再grep一下日志；

3. 多提供一种方法嘛。kill query是指你只是想停止这个语句，但是事务不会回滚。一般kill query是发生在客户端执行ctrl+c的时候啦。平时紧急处理确实直接用kill + thread_id。好问题

4. 对，另外，在kill query无效的时候，其实kill connection也是无效的

2019-01-26



Justin

0

想咨询一个问题 如果走索引找寻比如age=11的人的时候是只会锁age=10到age=12吗 如果那个索引页包含了从5到13的数据 是只会锁离11最近的还是说二分查找时候每一个访问到的都会锁

呢

2019-01-25

| 作者回复

只会锁左右。

2019-01-26



往事随风，顺其自然

0

12 号线程的等待逻辑是这样的：每 10 毫秒判断一下是否可以进入 InnoDB 执行，如果不行，如果不，就调用 nanosleep 函数进入 sleep 状态。这里为什么是 10 毫秒判断一下？怎么查看和设置这个参数？

2019-01-25



发条橙子。

0

老师我这里问一下唯一索引的问题，希望老师能给点思路

背景：一张商品库存表，如果表里没这个商品则插入，如果已经存在就更新库存。同步这个库存表是异步的，每次添加商品库存成功后会发消息，收到消息后会去表里新增/更新库存

问题：

商品库存表会有一个商品的唯一索引。

当我们批量添加同一商品库存后会批量发消息，消息同时生效后去处理就有了并发的问题。这时候两个消息都判断表里没有该商品记录，但是插入的时候就会有一个消息插入成功，另一个消息执行失败报唯一索引的错误，之后消息重试走更新的逻辑。

这个这样做对业务没有影响，但是现在批量添加的需求量上来了，线上一直报这种错误日志也不是个办法，我能想到的除了 catch 掉这个异常就没什么其他思路了。

老师能给一些其他的思路么

2019-01-25

| 作者回复

有唯一索引了，就直接插入，然后出现唯一性约束就放弃，这个逻辑的问题是啥，我感觉挺好的呀

是不是我没有 get 到问题的点

2019-01-25



AI杜嘉嘉

0

我想请问下老师，一个事务执行很长时间，我去 kill。那么，执行这个事务过程中的数据会不会回滚？

2019-01-25

| 作者回复

这个事务执行过程中新生成的数据吗？会回滚的

2019-01-25



曾剑

0



今天的问题，我觉得得让他自己执行完成后自动恢复。因为强制重启后该做的回滚还是会继续做。

2019-01-25



Dkey

0

老师，请教一个 第八章 的问题。关于可见性判断，文中都是说事务id大于高水位都不可见。如果等于是不是也不可见。还有一个， `readview`中是否不包含当前事务id。谢谢老师

2019-01-25

| 作者回复

代码实现上，事务生成`trxid`后，`trxid`的分配器会+1，以这个加1以后的数作为高水位，所以“等于”是不算的。

其实有没有包含是一样的，实现上没有包含。

2019-01-25

33 | 我查这么多数据，会不会把数据库内存打爆？

2019-01-28 林晓斌



我经常会被问到这样一个问题：我的主机内存只有100G，现在要对一个200G的大表做全表扫描，会不会把数据库主机的内存用光了？

这个问题确实值得担心，被系统OOM (out of memory) 可不是闹着玩的。但是，反过来想想，逻辑备份的时候，可不就是做整库扫描吗？如果这样就会把内存吃光，逻辑备份不是早就挂了？

所以说，对大表做全表扫描，看来应该是没问题的。但是，这个流程到底是怎么样的呢？

全表扫描对server层的影响

假设，我们现在要对一个200G的InnoDB表db1.t，执行一个全表扫描。当然，你要把扫描结果保存在客户端，会使用类似这样的命令：

```
mysql -h$host -P$port -u$user -p$pwd -e "select * from db1.t" > $target_file
```

你已经知道了，InnoDB的数据是保存在主键索引上的，所以全表扫描实际上是直接扫描表t的主键索引。这条查询语句由于没有其他的判断条件，所以查到的每一行都可以直接放到结果集里面，然后返回给客户端。

那么，这个“结果集”存在哪里呢？

实际上，服务端并不需要保存一个完整的结果集。取数据和发数据的流程是这样的：

1. 获取一行，写到net_buffer中。这块内存的大小是由参数net_buffer_length定义的，默认是16k。
2. 重复获取行，直到net_buffer写满，调用网络接口发出去。
3. 如果发送成功，就清空net_buffer，然后继续取下一行，并写入net_buffer。
4. 如果发送函数返回EAGAIN或WSAEWOULDBLOCK，就表示本地网络栈（socket send buffer）写满了，进入等待。直到网络栈重新可写，再继续发送。

这个过程对应的流程图如下所示。

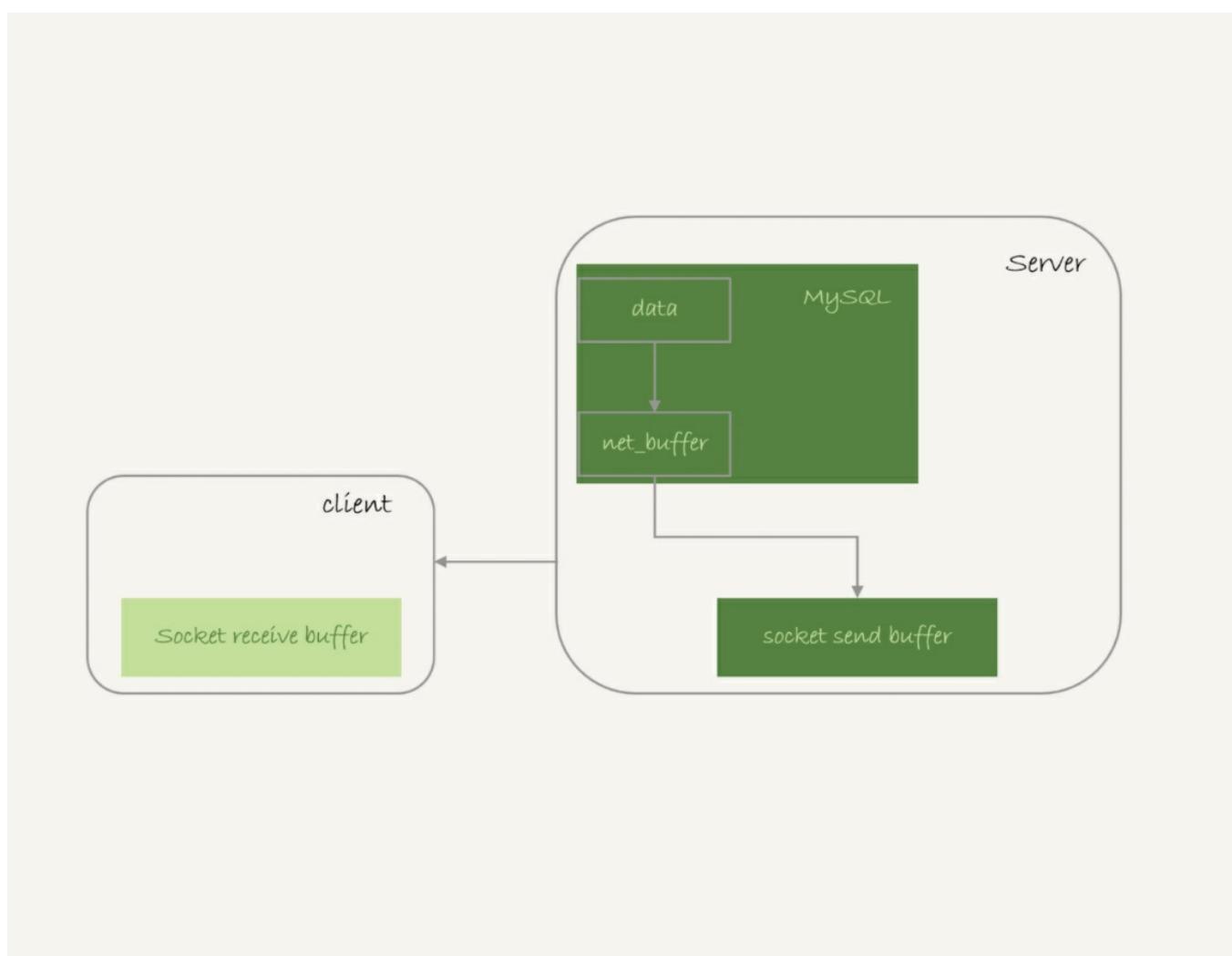


图1 查询结果发送流程

从这个流程中，你可以看到：

1. 一个查询在发送过程中，占用的MySQL内部的内存最大就是net_buffer_length这么大，并不会达到200G；
2. socket send buffer 也不可能达到200G（默认定义/proc/sys/net/core/wmem_default），如

果socket send buffer被写满，就会暂停读数据的流程。

也就是说，MySQL是“边读边发的”，这个概念很重要。这就意味着，如果客户端接收得慢，会导致MySQL服务端由于结果发不出去，这个事务的执行时间变长。

比如下面这个状态，就是我故意让客户端不去读socket receive buffer中的内容，然后在服务端show processlist看到的结果。

mysql> show processlist;						
Id	User	Host	db	Command	Time	State
4	root	localhost:61696	test	Query	0	starting
5	root	localhost:61772	test	Query	9	Sending to client

图2 服务端发送阻塞

如果你看到State的值一直处于“**Sending to client**”，就表示服务器端的网络栈写满了。

我在上一篇文章中曾提到，如果客户端使用-quick参数，会使用mysql_use_result方法。这个方法是读一行处理一行。你可以想象一下，假设有一个业务的逻辑比较复杂，每读一行数据以后要处理的逻辑如果很慢，就会导致客户端要过很久才会去取下一行数据，可能就会出现如图2所示的这种情况。

因此，对于正常的线上业务来说，如果一个查询的返回结果不会很多的话，我都建议你使用mysql_store_result这个接口，直接把查询结果保存到本地内存。

当然前提是查询返回结果不多。在[第30篇文章](#)评论区，有同学说到自己因为执行了一个大查询导致客户端占用内存近20G，这种情况下就需要改用mysql_use_result接口了。

另一方面，如果你在自己负责维护的MySQL里看到很多个线程都处于“**Sending to client**”这个状态，就意味着你要让业务开发同学优化查询结果，并评估这么多的返回结果是否合理。

而如果要快速减少处于这个状态的线程的话，将net_buffer_length参数设置为一个更大的值是一个可选方案。

与“**Sending to client**”长相很类似的一个状态是“**Sending data**”，这是一个经常被误会的问题。有同学问我说，在自己维护的实例上看到很多查询语句的状态是“**Sending data**”，但查看网络也没什么问题啊，为什么**Sending data**要这么久？

实际上，一个查询语句的状态变化是这样的（注意：这里，我略去了其他无关的状态）：

- MySQL查询语句进入执行阶段后，首先把状态设置成“**Sending data**”；
- 然后，发送执行结果的列相关的信息（meta data）给客户端；
- 再继续执行语句的流程；

- 执行完成后，把状态设置成空字符串。

也就是说，“**Sending data**”并不一定是指“正在发送数据”，而可能是处于执行器过程中的任意阶段。比如，你可以构造一个锁等待的场景，就能看到**Sending data**状态。

session A	session B
beign; select * from t where id=1 for update;	
	select * from t lock in share mode; (blocked)

图3 读全表被锁

mysql> show processlist;							
Id	User	Host	db	Command	Time	State	Info
4	root	localhost:15392	test	Sleep	59		NULL
5	root	localhost:15406	test	Query	3	Sending data	select * from t lock in share mode
9	root	localhost:16412	test	Query	0	starting	show processlist

图 4 Sending data状态

可以看到，**session B**明显是在等锁，状态显示为**Sending data**。

也就是说，仅当一个线程处于“等待客户端接收结果”的状态，才会显示“**Sending to client**”；而如果显示成“**Sending data**”，它的意思只是“正在执行”。

现在你知道了，查询的结果是分段发给客户端的，因此扫描全表，查询返回大量的数据，并不会把内存打爆。

在**server**层的处理逻辑我们都清楚了，在**InnoDB**引擎里面又是怎么处理的呢？扫描全表会不会对引擎系统造成影响呢？

全表扫描对**InnoDB**的影响

在[第2](#)和[第15](#)篇文章中，我介绍**WAL**机制的时候，和你分析了**InnoDB**内存的一个作用，是保存更新的结果，再配合**redo log**，就避免了随机写盘。

内存的数据页是在**Buffer Pool (BP)**中管理的，在**WAL**里**Buffer Pool**起到了加速更新的作用。而实际上，**Buffer Pool**还有一个更重要的作用，就是加速查询。

在第2篇文章的评论区有同学问道，由于有**WAL**机制，当事务提交的时候，磁盘上的数据页是旧的，那如果这时候马上有一个查询要来读这个数据页，是不是要马上把**redo log**应用到数据页呢？

答案是不需要。因为这时候内存数据页的结果是最新的，直接读内存页就可以了。你看，这时候查询根本不需要读磁盘，直接从内存拿结果，速度是很快的。所以说，**Buffer Pool**还有加速查询的作用。

而**Buffer Pool**对查询的加速效果，依赖于一个重要的指标，即：**内存命中率**。

你可以在**show engine innodb status**结果中，查看一个系统当前的BP命中率。一般情况下，一个稳定服务的线上系统，要保证响应时间符合要求的话，内存命中率要在**99%**以上。

执行**show engine innodb status**，可以看到“**Buffer pool hit rate**”字样，显示的就是当前的命中率。比如图5这个命中率，就是**99.0%**。



Buffer pool hit rate 990 / 1000

图5 **show engine innodb status**显示内存命中率

如果所有查询需要的数据页都能够直接从内存得到，那是最好的，对应的命中率就是**100%**。但，这在实际生产上是很难做到的。

InnoDB Buffer Pool的大小是由参数 **innodb_buffer_pool_size**确定的，一般建议设置成可用物理内存的**60%~80%**。

在大约十年前，单机的数据量是上百个G，而物理内存是几个G；现在虽然很多服务器都能有**128G**甚至更高的内存，但是单机的数据量却达到了T级别。

所以，**innodb_buffer_pool_size**小于磁盘的数据量是很常见的。如果一个 Buffer Pool满了，而又要从磁盘读入一个数据页，那肯定是要淘汰一个旧数据页的。

InnoDB内存管理用的是最近最少使用 (**Least Recently Used, LRU**)算法，这个算法的核心就是淘汰最久未使用的数据。

下图是一个LRU算法的基本模型。

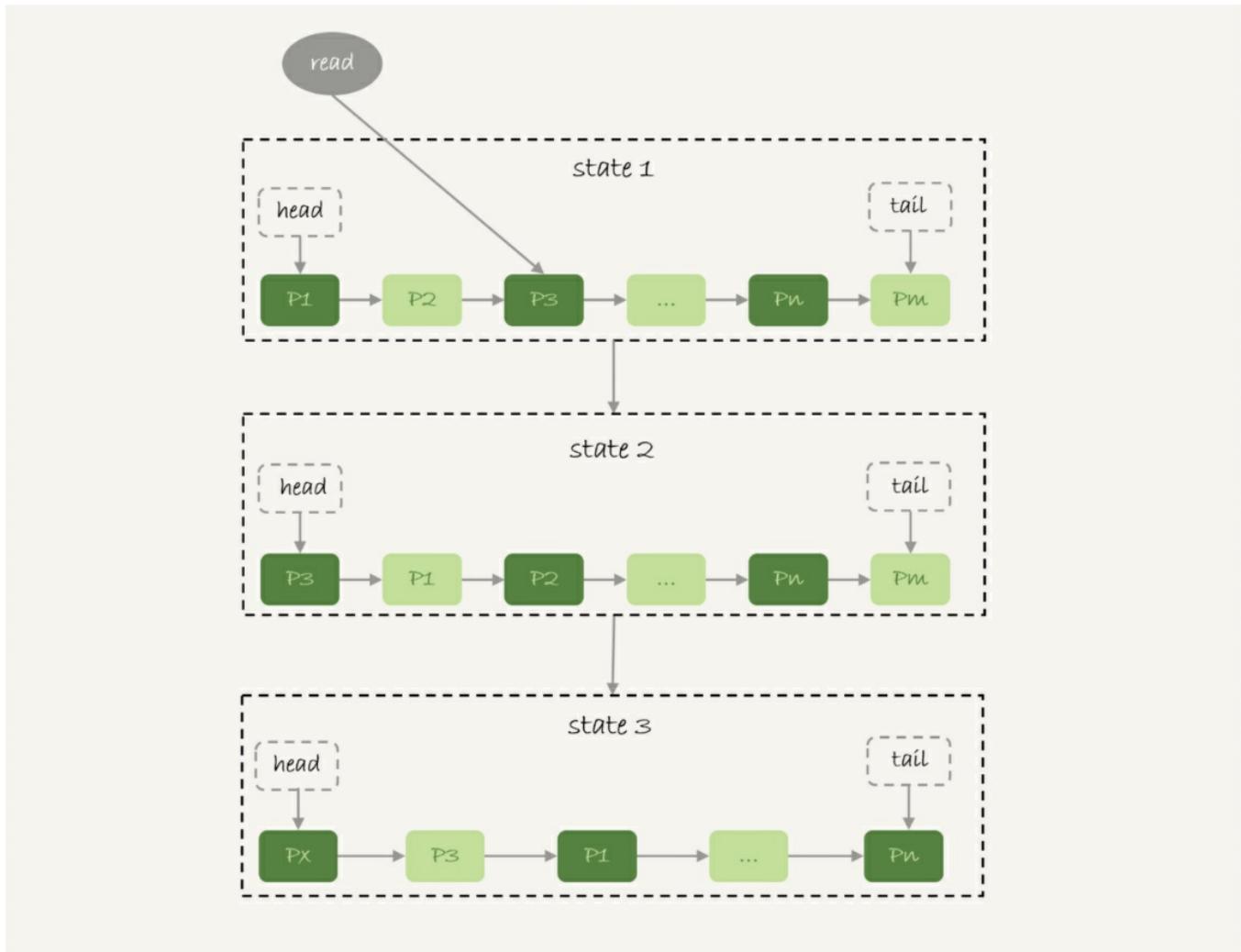


图6 基本LRU算法

InnoDB管理Buffer Pool的LRU算法，是用链表来实现的。

1. 在图6的状态1里，链表头部是P1，表示P1是最近刚刚被访问过的数据页；假设内存里只能放下这么多数据页；
2. 这时候有一个读请求访问P3，因此变成状态2，P3被移到最前面；
3. 状态3表示，这次访问的数据页是不存在于链表中的，所以需要在Buffer Pool中新申请一个数据页Px，加到链表头部。但是由于内存已经满了，不能申请新的内存。于是，会清空链表末尾Pm这个数据页的内存，存入Px的内容，然后放到链表头部。
4. 从效果上看，就是最久没有被访问的数据页Pm，被淘汰了。

这个算法乍一看上去没什么问题，但是如果考虑到要做一个全表扫描，会不会有问题呢？

假设按照这个算法，我们要扫描一个200G的表，而这个表是一个历史数据表，平时没有业务访问它。

那么，按照这个算法扫描的话，就会把当前的Buffer Pool里的数据全部淘汰掉，存入扫描过程中

访问到的数据页的内容。也就是说Buffer Pool里面主要放的是这个历史数据表的数据。

对于一个正在做业务服务的库，这可不妙。你会看到，Buffer Pool的内存命中率急剧下降，磁盘压力增加，SQL语句响应变慢。

所以，InnoDB不能直接使用这个LRU算法。实际上，InnoDB对LRU算法做了改进。



图 7 改进的LRU算法

在InnoDB实现上，按照5:3的比例把整个LRU链表分成了young区域和old区域。图中LRU_old指向的就是old区域的第一个位置，是整个链表的5/8处。也就是说，靠近链表头部的5/8是young区域，靠近链表尾部的3/8是old区域。

改进后的LRU算法执行流程变成了下面这样。

1. 图7中状态1，要访问数据页 P_3 ，由于 P_3 在young区域，因此和优化前的LRU算法一样，将其移到链表头部，变成状态2。
2. 之后要访问一个新的不存在于当前链表的数据页，这时候依然是淘汰掉数据页 P_m ，但是新插入的数据页 P_x ，是放在LRU_old处。

3. 处于**old**区域的数据页，每次被访问的时候都要做下面这个判断：

- 若这个数据页在**LRU**链表中存在的时间超过了1秒，就把它移动到链表头部；
- 如果这个数据页在**LRU**链表中存在的时间短于1秒，位置保持不变。1秒这个时间，是由参数**innodb_old_blocks_time**控制的。其默认值是**1000**，单位毫秒。

这个策略，就是为了处理类似全表扫描的操作量身定制的。还是以刚刚的扫描**200G**的历史数据表为例，我们看看改进后的**LRU**算法的操作逻辑：

1. 扫描过程中，需要新插入的数据页，都被放到**old**区域；
2. 一个数据页里面有多条记录，这个数据页会被多次访问到，但由于是顺序扫描，这个数据页第一次被访问和最后一次被访问的时间间隔不会超过1秒，因此还是会保留在**old**区域；
3. 再继续扫描后续的数据，之前的这个数据页之后也不会再被访问到，于是始终没有机会移到链表头部（也就是**young**区域），很快就会被淘汰出去。

可以看到，这个策略最大的收益，就是在扫描这个大表的过程中，虽然也用到了**Buffer Pool**，但是对**young**区域完全没有影响，从而保证了**Buffer Pool**响应正常业务的查询命中率。

小结

今天，我用“大查询会不会把内存用光”这个问题，和你介绍了**MySQL**的查询结果，发送给客户端的过程。

由于**MySQL**采用的是边算边发的逻辑，因此对于数据量很大的查询结果来说，不会在**server**端保存完整的结果集。所以，如果客户端读结果不及时，会堵住**MySQL**的查询过程，但是不会把内存打爆。

而对于**InnoDB**引擎内部，由于有淘汰策略，大查询也不会导致内存暴涨。并且，由于**InnoDB**对**LRU**算法做了改进，冷数据的全表扫描，对**Buffer Pool**的影响也能做到可控。

当然，我们前面文章有说过，全表扫描还是比较耗费**IO**资源的，所以业务高峰期还是不能直接在线上主库执行全表扫描的。

最后，我给你留一个思考题吧。

我在文章中说到，如果由于客户端压力太大，迟迟不能接收结果，会导致**MySQL**无法发送结果而影响语句执行。但，这还不是最糟糕的情况。

你可以设想出由于客户端的性能问题，对数据库影响更严重的例子吗？或者你是否经历过这样的场景？你又是怎么优化的？

你可以把你的经验和分析写在留言区，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收

听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，如果一个事务被kill之后，持续处于回滚状态，从恢复速度的角度看，你是应该重启等它执行结束，还是应该强行重启整个MySQL进程。

因为重启之后该做的回滚动作还是不能少的，所以从恢复速度的角度来说，应该让它自己结束。

当然，如果这个语句可能会占用别的锁，或者由于占用IO资源过多，从而影响到了别的语句执行的话，就需要先做主备切换，切到新主库提供服务。

切换之后别的线程都断开了连接，自动停止执行。接下来还是等它自己执行完成。这个操作属于我们在文章中说到的，减少系统压力，加速终止逻辑。

评论区留言点赞板：

@HuaMax 的回答中提到了对其他线程的影响；
@夹心面包 @Ryoma @曾剑 同学提到了重启后依然继续做回滚操作的逻辑。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Lin Xiaobin, a man with short dark hair and glasses, wearing a black button-down shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45 讲' in large, bold, dark font, with the subtitle '从原理到实战，丁奇带你搞懂 MySQL' below it. To the far left is the '极客时间' logo. Below the title, there's a section for the instructor: '林晓斌' (Lin Xiaobin), '网名丁奇' (alias Dingqi), and '前阿里资深技术专家' (Former senior technical expert at Alibaba). At the bottom, there's a call-to-action: '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。' (New version upgrade: Click 'Invite friends to read', get 10 free reads, and there are cash rewards for subscriptions).

精选留言



700

老师，您好。根据文章内容，提炼如下信息：

2

如果你看到 State 的值一直处于“**Sending to client**”，就表示服务器端的网络栈写满了。

如何处理？

1) 使用 `mysql_store_result` 这个接口，直接把查询结果保存到本地内存。

2) 优化查询结果，并评估这么多的返回结果是否合理。

3) 而如果要快速减少处于这个状态的线程的话，将 `net_buffer_length` 参数设置为一个更大的值是一个可选方案。

对于第3)方案不是很懂，“**Sending to client**” 表示服务器端的网路栈写满了，那不是应该加大 `socket send buffer` 吗？跟加大 `net_buffer_length` 有什么关系？`net_buffer_length` 加再大，但 `socket send buffer` 很小的话，网络栈不还是处于写满状态？

2019-01-28

| 作者回复

好问题| 很好的思考|

是这样的，`net_buffer_length` 的最大值是 **1G**，这个值比 `socket send buffer` 大（一般是几**M**）

比如假设一个业务，他的平均查询结果都是 **10M**（当然这个业务有问题，最终是要通过业务解决）

但是如果我把 `net_buffer_length` 改成 **10M**，就不会有“**Sending to client**”的情况。虽然网络栈还是慢慢发的，但是那些没发完的都缓存在 `net_buffer` 中，对于执行器来说，都是“已经写出去了”。

2019-01-28



Long

4

最近没时间看，今天终于补完了几天的课。

2019-01-28



长杰

3

遇到过一个场景，用 `mysqldump` 对业务 `db` 做逻辑备份保存在客户端，客户端是虚拟机，磁盘很快满了，导致 `server` 端出现 `sending to client` 状态，更糟糕的是业务 `db` 更新频繁，导致 `undo` 表空间变大，`db` 服务堵塞，服务端磁盘空间不足。

2019-01-28

| 作者回复

非常好，正是我要说明的一个场景呢，直接用你的例子放在下篇答疑部分哈

2019-01-29



Sinyo

1

@700 的置顶提问

老师你说：“但是如果把 `net_buffer_length` 改成 **10M**，就不会有“**Sending to client**”的情况。虽然网络栈还是慢慢发的，但是那些没发完的都缓存在 `net_buffer` 中，对于执行器来说，都是“已经写出去了”。”

假如数据量有1G，而如果要快速减少处于这个状态的线程的话，我们把net_buffer_length从10M改成1G，快速减少的那部分操作是不是只有服务端发送到net_buffer的这部分？这样就不会有“Sending to client”的情况么？

2019-01-29

作者回复

还是会显示为“Sending to client”，但是语句已经执行完了。

不会占着资源（比如MDL读锁）

2019-01-29



700

1

老师，您好。感谢解答。

接上个问题。

Sending to client 是发生在下面哪个阶段的事件呢？

1)是“获取一行，写到 net_buffer 中。”

2)还是“直到 net_buffer 写满，调用网络接口发出去。”//即数据从 net_buffer 发到 socket send buffer？

3)还是“将 socket send buffer 的数据发送给 socket receive buffer”

从您的回答“但是如果我把net_buffer_length 改成10M，就不会有“Sending to client”的情况。”，我感觉应该是属于第1)阶段的事件。但感觉这又与您说的“Sending to client 表示的是服务器端的网络栈写满了”相矛盾。

2019-01-28

作者回复

写net_buffer --> net_buffer满了，调用网络接口发 -->发不出去

这个是同一个调用链条呀

“哪个阶段”没看懂，是同一个时刻

2019-01-28



慕塔

0

young区域其实还有优化，频道调整LRU页的顺序为影响性能(LRU很长)，如果要读页在young区域某位置，其实是没有必要将要读页拿到头部，本身已在热点区。页的属性有一个时间戳字段，可以用于计算处于old区域的时间。』

2019-02-03



Mr.Strive.Z.H.L

0

老师您好：

我看到评论的问题，有个疑惑：

“

之前有特殊功能需要从主要业务库拉取指定范围的数据到另外同一个库的其他数据表的动作（insert into xxxx select xxx from xxx 这种操作）数据量在万级或者十万级，对于这种操作，和本文讲的应该有些不同吧？能否帮分析一下这种场景的大致情况呢？或者有什么好的建议吗？

作者回复: 嗯, 这个不会返回结果到客户端, 所以网络上不会有

引擎内部的扫描机制是差不多的

唯一不同是这个过程可能对原表有行锁 (如果设置的是RR)

万或者十万还好, 是小数据, 可以考虑拿到客户端再写回去, 避免锁的问题

”

先把数据拿回客户端, 再insert到另一个库。是为了避免锁的问题。

这里从原库拉取数据就是select语句, 没有涉及到next-key锁呀, 为啥会有锁的问题呢?

2019-02-01

| 作者回复

好问题[], 第40篇会说这个问题哈, 新年快乐

2019-02-01



梁中华

0

感觉young 和old 的叫法反了, 后面的应该叫young 才好理解。另外文中的old 区也会有类似young 区域的淘汰策略吧

2019-01-30

| 作者回复

好几个同学这么说, 我都方了[]

这句是官方文档上的

“Accessing a page in the old sublist makes it “young”, moving it to the head of the buffer pool”

2019-01-30



Leon[]

0

如果客户端读结果不及时, 会堵住 MySQL 的查询过程, 但是不会把内存打爆。这个是指客户端的tcp滑动窗口处理没有及时确认, 导致server端的网络协议栈没有多余的空间可以发送数据, 导致server的处理线程停止从db读取数据发给client, 是这样理解吗

2019-01-30

| 作者回复

对的

2019-01-30



Richie

0

老师, 怎么才能了解什么地方占用内存, 查了很多资料都没有这方面的信息, MySQL5.6

2019-01-30

| 作者回复

这个官方版本确实是还没有系统的地方查看~

2019-02-04



changshan

0

老师好，咨询一个于之前文章有关的问题，在rr隔离级别下会产生幻读，然而这个幻读mysql是通过什么机制来解决的呢？有的说是mvcc，有的说是next-key锁。有点疑惑了。另外，怎么能够验证mysql使用具体的哪种技术解决了幻读？

2019-01-29

| 作者回复

看一下20和21篇哈

2019-01-29



天使梦泪

0

老师好，针对我上次问您的mysql缓存中的数据储存问题，您回答可以一直保存的，具体是怎么实现一直保存的（也不是储存在磁盘上，是使用的内存）？内存重启了之后，缓存不就也丢失了，是怎么做到持久化保存的，老师可以帮忙详细解答下么？

2019-01-29

| 作者回复

InnoDB 的是buffer pool，是在内存里。

“内存重启了之后，缓存不就也丢失了，是怎么做到持久化保存的，老师可以帮忙详细解答下么？”

没有保存，重启就没有了，要访问的时候需要重新去磁盘读

2019-01-29



有铭

0

感觉mysql的做法有点流式读取的意思。

但是，老师，虽然这篇文章讲述了Mysql是如何“边读边发”。但是更复杂的情况没有说明，比如我现在要执行一个复杂的查询，而且查询是排序的，这意味着mysql需要对整个结构排序，然后才能一条条的发出去，如果数据量极大的情况，Mysql如何完成排序过程，需要把数据全部载入内存吗？还是存储在缓存文件里搞分而治之的排序，然后再“边读边发”

2019-01-29

| 作者回复

看一下 <https://time.geekbang.org/column/article/73479> 这篇文章的图5哈

有说到哦

2019-01-29



Max

0

林Sir,你好。

曾经发生过二个问题

第一个问题是show columns from table带来的临时表产生和移除

大量的session opening tmp tables 和removing tmp tables

也kill不掉会话，首先主从先切，让原主停止对外服务。在kill掉所有用户会话。

问题解决，同时修改innodb_thread_concurrency参数数量。

另外一个感觉是mysql bug引起的。

当时环境是percona-mysql-20-21主从环境

没有高并发所，所有的用户会话状态都是query end，会话不释放。

造成会话连接数暴涨。撑满了所有的会话。

查看engine innodb status，发现latch等待非常高

OS WAIT ARRAY INFO: signal count 5607657

RW-shared spins 0, rounds 2702261, OS waits 70377

RW-excl spins 0, rounds 216191633, OS waits 1802414

RW-sx spins 1588, rounds 5965, OS waits 70

Spin rounds per wait: 2702261.00 RW-shared, 216191633.00 RW-excl, 3.76 RW-sx

MySQL thread id 79467374, OS thread handle 140327525230336, query id 949505008 10.0.2.6

apirwuser query end

INSERT INTO `xxxxxx` ('xxxx','xxxx','xxxx','xxxx') VALUES ('c2aab326-adf9-470b-940e-133fa2c7f685','android','862915033153129',1535597836)

---TRANSACTION 1154797559, ACTIVE (PREPARED) 1 sec

mysql tables in use 1, locked 1

第二个问题一直没有解决，后来把mysql 5.7 降到mysql 5.6

还有一个关于out of memory问题

sql如下: a是父表， b是子表

select a.id,a.name,b.title from a inner join b on a.id=b.aid

where a.create_time>'2010-08-01 00:00:00' and a.create_time<'2010-08-10 23:59:59'

它的执行计划应该是

1 a表和b表生产迪卡集产生虚列集合T。2从集合T筛选出 a.id(主键)=b.aid(外键)产生虚集合V3最后从集合v筛选出where条件，得到最终结果。

如果二个表都超过千万条记录，产生的集合数据应该是放到内存中。如果是这样会不会打暴内存

2019-01-29

| 作者回复

1. 是的， show columns 其实不是一个好操作

2. 这个没见过，也没印象在社区中碰到这种现象，降成5.6就好了是吗？

3. 不会的， 34、35两篇就是说这个问题的哈

2019-01-31



PHP-SICUN

0

老师，您好，有两个问题麻烦解惑一下

1.扫描200G的表时数据会先放到InnoDB buffer pool,然后发送时在读取到net_buffer吗？

2.如果是的话，异常情况导致socket send buffer被写满，是不是会出现InnoDB buffer中的某一页有可能出现读取后面的行时，超过1s，而被放到yong区域的情况？

不知道这样表述或者理解的对吗

2019-01-29

| 作者回复

1. 是，但是也不是“全部放到buffer pool以后”才发，读的时候是一个page一个page地读的
2. 会，在这个是“某一页”而已，量不大。好问题

很好的思考

2019-01-29



Ryoma

0

有两个问题：

0: MySQL 中的新生代和老生代的名字这么反人类的么

1: 我在使用**show engine innodb status**看Buffer Pool时，发现Buffer Pool有8个（0~8），请问老师这个是什么策略呢？

2019-01-28

| 作者回复

0

1. 搜一下 **innodb_buffer_pool_instances** 这个参数的解释哈

2019-01-28



老杨同志

0

老师提示考虑两个客户端都进行**update**的情况。

如果第一个客户端执行**select * from t for update** 而迟迟不读取返回的数据，会造成**server**端长期占用记录的行锁，如果其他线程要更新被锁定的记录，会报锁等待超时的错误

2019-01-28

| 作者回复

0

2019-01-28



天使梦泪

0

老师，我有个问题不明白，mysql从缓存中取数据，缓存里的数据是怎么实现可以保存一段时间的？

2019-01-28

| 作者回复

“保存一段时间”是啥意思，**LRU**算法不是按照时间的哈，如果没人来淘汰，是可以一直保存的。

。

2019-01-28



如明如月

0

之前有特殊功能需要从主要业务库拉取指定范围的数据到另外同一个库的其他数据表的动作 (**insert into xxxx select xxx from xxx** 这种操作) 数据量在万级或者十万级，对于这种操作，和本文讲的应该有些不同吧？能否帮分析一下这种场景的大致情况呢？或者有什么好的建议吗？

2019-01-28

| 作者回复

嗯，这个不会返回结果到客户端，所以网络上不会有问题是

引擎内部的扫描机制是差不多的

唯一不同是这个过程可能对原表有行锁（如果设置的是RR）

万或者十万还好，是小数据，可以考虑拿到客户端再写回去，避免锁的问题

2019-01-28



garming

0

老师你好，如果是MyISAM存储引擎，大查询会导致内存暴涨吗？如果过，是什么原因呢？

2019-01-28

| 作者回复

也是不会的，跟InnoDB一样

2019-01-28

34 | 到底可不可以使用join?

2019-01-30 林晓斌



在实际生产中，关于**join**语句使用的问题，一般会集中在以下两类：

1. 我们DBA不让使用**join**，使用**join**有什么问题呢？
2. 如果有两个大小不同的表做**join**，应该用哪个表做驱动表呢？

今天这篇文章，我就先跟你说说**join**语句到底是怎么执行的，然后再来回答这两个问题。

为了便于量化分析，我还是创建两个表t1和t2来和你说明。

```
CREATE TABLE `t2` (
  `id` int(11) NOT NULL,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `a` (`a`)
) ENGINE=InnoDB;
```

```
drop procedure idata;
delimiter ;;
create procedure idata()
begin
  declare i int;
  set i=1;
  while(i<=1000)do
    insert into t2 values(i, i, i);
    set i=i+1;
  end while;
end;;
delimiter ;
call idata();

create table t1 like t2;
insert into t1 (select * from t2 where id<=100)
```

可以看到，这两个表都有一个主键索引**id**和一个索引**a**，字段**b**上无索引。存储过程**idata()**往表**t2**里插入了1000行数据，在表**t1**里插入的是100行数据。

Index Nested-Loop Join

我们来看一下这个语句：

```
select * from t1 straight_join t2 on (t1.a=t2.a);
```

如果直接使用**join**语句，MySQL优化器可能会选择表**t1**或**t2**作为驱动表，这样会影响我们分析SQL语句的执行过程。所以，为了便于分析执行过程中的性能问题，我改用**straight_join**让MySQL使用固定的连接方式执行查询，这样优化器只会按照我们指定的方式去**join**。在这个语句

里，**t1**是驱动表，**t2**是被驱动表。

现在，我们来看一下这条语句的**explain**结果。

```
mysql> explain select * from t1 straight_join t2 on (t1.a=t2.a);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | NULL | ALL | a | NULL | NULL | NULL | 100 | 100.00 | Using where |
| 1 | SIMPLE | t2 | NULL | ref | a | a | 5 | test.t1.a | 1 | 100.00 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图1 使用索引字段join的 explain结果

可以看到，在这条语句里，被驱动表**t2**的字段**a**上有索引，**join**过程用上了这个索引，因此这个语句的执行流程是这样的：

1. 从表**t1**中读入一行数据 **R**;
2. 从数据行**R**中，取出**a**字段到表**t2**里去查找;
3. 取出表**t2**中满足条件的行，跟**R**组成一行，作为结果集的一部分;
4. 重复执行步骤1到3，直到表**t1**的末尾循环结束。

这个过程是先遍历表**t1**，然后根据从表**t1**中取出的每行数据中的**a**值，去表**t2**中查找满足条件的记录。在形式上，这个过程就跟我们写程序时的嵌套查询类似，并且可以用上被驱动表的索引，所以我们称之为“**Index Nested-Loop Join**”，简称**NLJ**。

它对应的流程图如下所示：

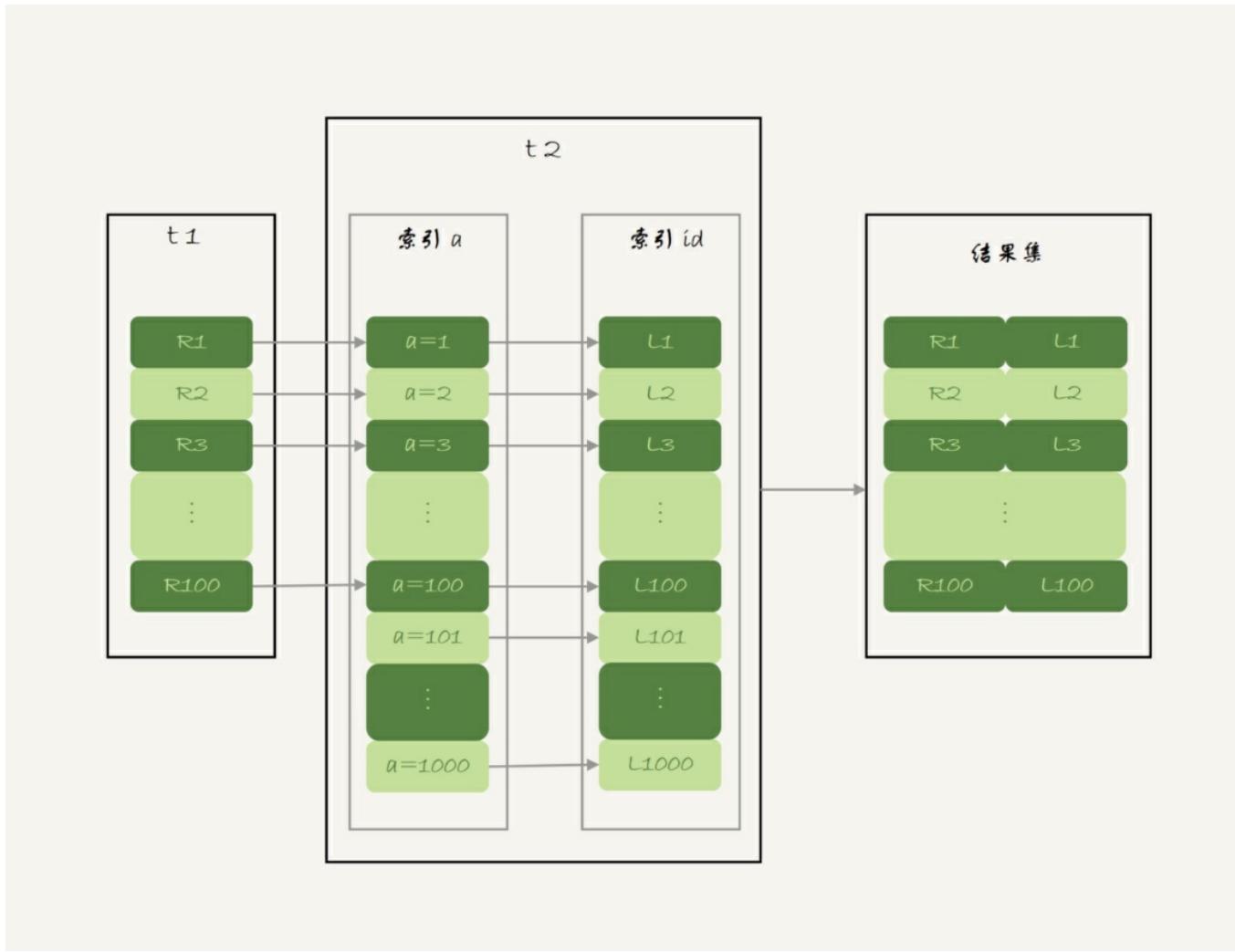


图2 Index Nested-Loop Join算法的执行流程

在这个流程里：

1. 对驱动表t1做了全表扫描，这个过程需要扫描100行；
2. 而对于每一行R，根据a字段去表t2查找，走的是树搜索过程。由于我们构造的数据都是一一对应的，因此每次的搜索过程都只扫描一行，也是总共扫描100行；
3. 所以，整个执行流程，总扫描行数是200。

现在我们知道了这个过程，再试着回答一下文章开头的两个问题。

先看第一个问题：能不能使用**join**？

假设不使用**join**，那我们就只能用单表查询。我们看看上面这条语句的需求，用单表查询怎么实现。

1. 执行**select * from t1**，查出表t1的所有数据，这里有100行；
2. 循环遍历这100行数据：

- 从每一行**R**取出字段**a**的值\$R.a;
- 执行select * from t2 where a=\$R.a;
- 把返回的结果和**R**构成结果集的一行。

可以看到，在这个查询过程，也是扫描了200行，但是总共执行了101条语句，比直接join多了100次交互。除此之外，客户端还要自己拼接SQL语句和结果。

显然，这么做还不如直接join好。

我们再来看看第二个问题：怎么选择驱动表？

在这个join语句执行过程中，驱动表是走全表扫描，而被驱动表是走树搜索。

假设被驱动表的行数是M。每次在被驱动表查一行数据，要先搜索索引a，再搜索主键索引。每次搜索一棵树近似复杂度是以2为底的M的对数，记为 $\log_2 M$ ，所以在被驱动表上查一行的时间复杂度是 $2 * \log_2 M$ 。

假设驱动表的行数是N，执行过程就要扫描驱动表N行，然后对于每一行，到被驱动表上匹配一次。

因此整个执行过程，近似复杂度是 $N + N * 2 * \log_2 M$ 。

显然，N对扫描行数的影响更大，因此应该让小表来做驱动表。

如果你没觉得这个影响有那么“显然”，可以这么理解：N扩大1000倍的话，扫描行数就会扩大1000倍；而M扩大1000倍，扫描行数扩大不到10倍。

到这里小结一下，通过上面的分析我们得到了两个结论：

1. 使用join语句，性能比强行拆成多个单表执行SQL语句的性能要好；
2. 如果使用join语句的话，需要让小表做驱动表。

但是，你需要注意，这个结论的前提是“可以使用被驱动表的索引”。

接下来，我们再看看被驱动表用不上索引的情况。

Simple Nested-Loop Join

现在，我们把SQL语句改成这样：

```
select * from t1 straight_join t2 on (t1.a=t2.b);
```

由于表t2的字段b上没有索引，因此再用图2的执行流程时，每次到t2去匹配的时候，就要做一次

全表扫描。

你可以先设想一下这个问题，继续使用图2的算法，是不是可以得到正确的结果呢？如果只看结果的话，这个算法是正确的，而且这个算法也有一个名字，叫做“**Simple Nested-Loop Join**”。

但是，这样算来，这个**SQL**请求就要扫描表t2多达100次，总共扫描 $100 \times 1000 = 10$ 万行。

这还只是两个小表，如果t1和t2都是10万行的表（当然了，这也还是属于小表的范围），就要扫描100亿行，这个算法看上去太“笨重”了。

当然，MySQL也没有使用这个**Simple Nested-Loop Join**算法，而是使用了另一个叫作“**Block Nested-Loop Join**”的算法，简称BNL。

Block Nested-Loop Join

这时候，被驱动表上没有可用的索引，算法的流程是这样的：

1. 把表t1的数据读入线程内存join_buffer中，由于我们这个语句中写的是**select ***，因此是把整个表t1放入了内存；
2. 扫描表t2，把表t2中的每一行取出来，跟join_buffer中的数据做对比，满足join条件的，作为结果集的一部分返回。

这个过程的流程图如下：

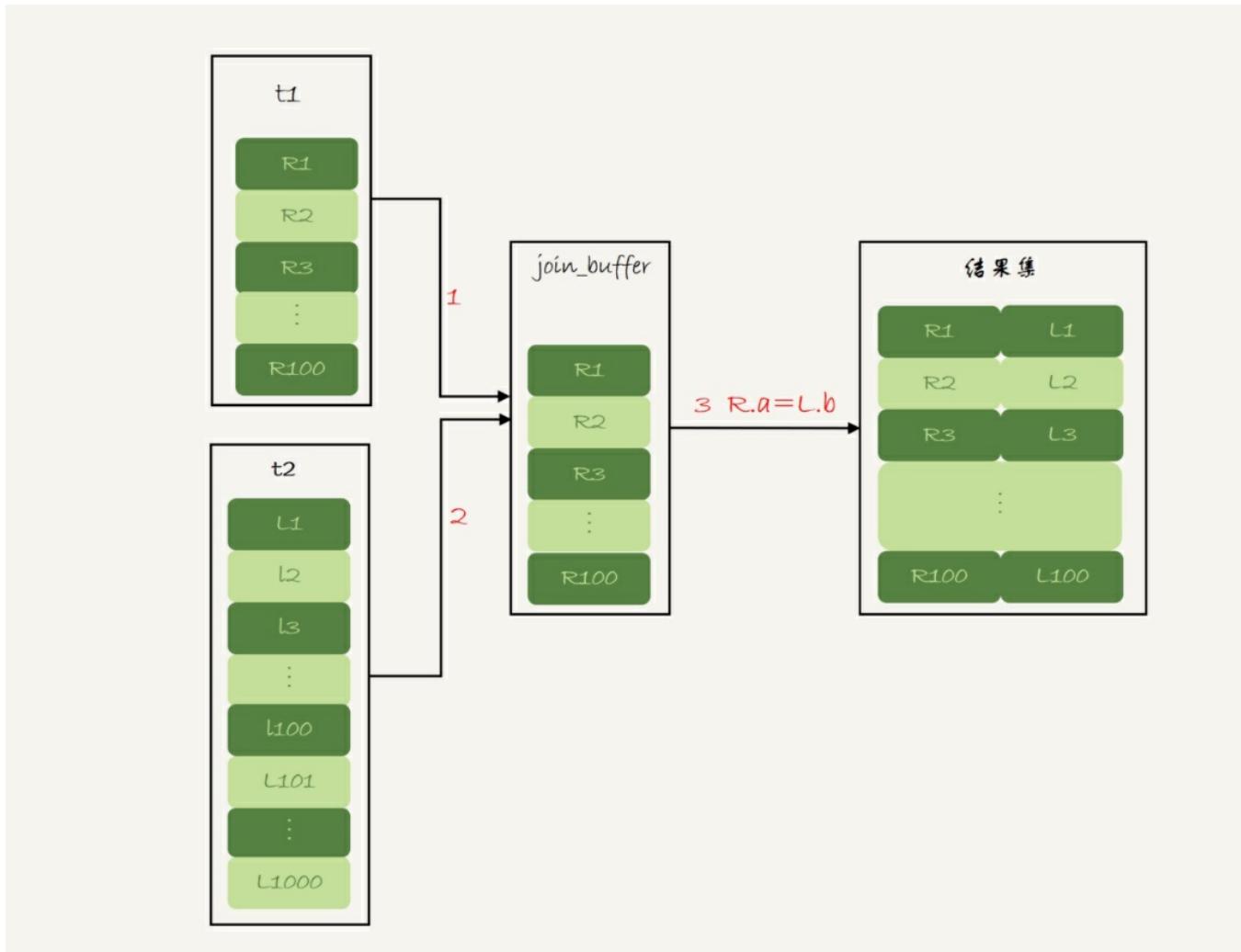


图3 Block Nested-Loop Join 算法的执行流程

对应地，这条SQL语句的explain结果如下所示：

```
mysql> explain select * from t1 straight_join t2 on (t1.a=t2.b);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | t1   | NULL      | ALL  | a             | NULL | NULL    | NULL | 100  | 100.00    | NULL
| 1 | SIMPLE     | t2   | NULL      | ALL  | NULL          | NULL | NULL    | NULL | 1000 | 10.00     | Using where; Using join buffer (Block Nested Loop)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图4 不使用索引字段join的 explain结果

可以看到，在这个过程中，对表**t1**和**t2**都做了一次全表扫描，因此总的扫描行数是1100。由于**join_buffer**是以无序数组的方式组织的，因此对表**t2**中的每一行，都要做100次判断，总共需要在内存中做的判断次数是： $100 \times 1000 = 10$ 万次。

前面我们说过，如果使用**Simple Nested-Loop Join**算法进行查询，扫描行数也是10万行。因此，从时间复杂度上来说，这两个算法是一样的。但是，**Block Nested-Loop Join**算法的这10万次判断是内存操作，速度上会快很多，性能也更好。

接下来，我们来看一下，在这种情况下，应该选择哪个表做驱动表。

假设小表的行数是**N**，大表的行数是**M**，那么在这个算法里：

1. 两个表都做一次全表扫描，所以总的扫描行数是M+N；
2. 内存中的判断次数是M*N。

可以看到，调换这两个算式中的M和N没差别，因此这时候选择大表还是小表做驱动表，执行耗时是一样的。

然后，你可能马上就会问了，这个例子里表t1才100行，要是表t1是一个大表，join_buffer放不下怎么办呢？

join_buffer的大小是由参数join_buffer_size设定的，默认值是256k。如果放不下表t1的所有数据话，策略很简单，就是分段放。我把join_buffer_size改成1200，再执行：

```
select * from t1 straight_join t2 on (t1.a=t2.b);
```

执行过程就变成了：

1. 扫描表t1，顺序读取数据行放入join_buffer中，放完第88行join_buffer满了，继续第2步；
2. 扫描表t2，把t2中的每一行取出来，跟join_buffer中的数据做对比，满足join条件的，作为结果集的一部分返回；
3. 清空join_buffer；
4. 继续扫描表t1，顺序读取最后的12行数据放入join_buffer中，继续执行第2步。

执行流程图也就变成这样：

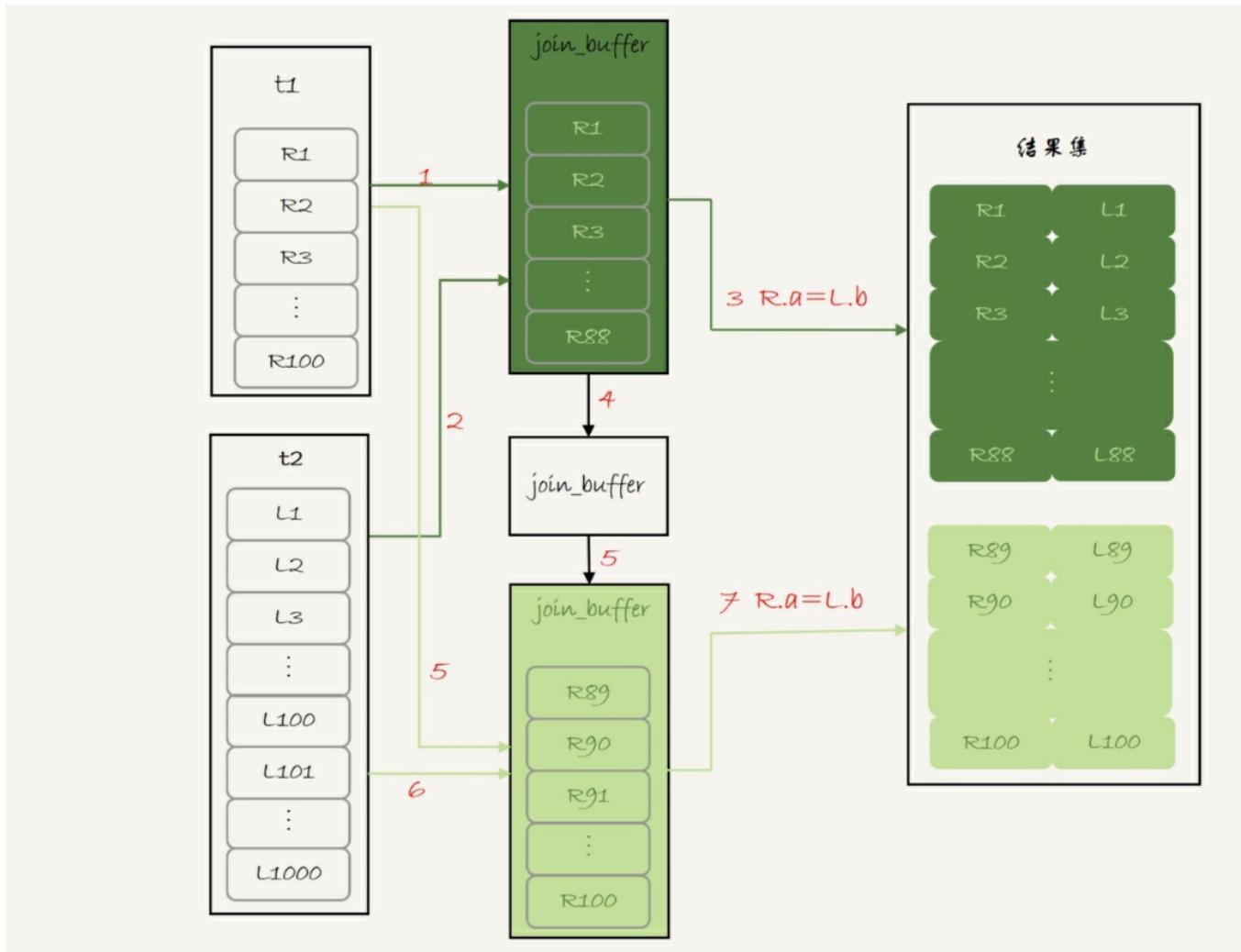


图5 Block Nested-Loop Join – 两段

图中的步骤4和5，表示清空join_buffer再复用。

这个流程才体现出了这个算法名字中“**Block**”的由来，表示“分块去join”。

可以看到，这时候由于表 t_1 被分成了两次放入join_buffer中，导致表 t_2 会被扫描两次。虽然分成两次放入join_buffer，但是判断等值条件的次数还是不变的，依然是 $(88+12)*1000=10$ 万次。

我们再来看下，在这种情况下驱动表的选择问题。

假设，驱动表的数据行数是 N ，需要分 K 段才能完成算法流程，被驱动表的数据行数是 M 。

注意，这里的 K 不是常数， N 越大 K 就会越大，因此把 K 表示为 $\lambda * N$ ，显然 λ 的取值范围是 $(0,1)$ 。

所以，在这个算法的执行过程中：

1. 扫描行数是 $N + \lambda * N * M$;
2. 内存判断 $N * M$ 次。

显然，内存判断次数是不受选择哪个表作为驱动表影响的。而考虑到扫描行数，在 M 和 N 大小确

定的情况下， N 小一些，整个算式的结果会更小。

所以结论是，应该让小表当驱动表。

当然，你会发现，在 $N+\lambda \cdot N \cdot M$ 这个式子里， λ 才是影响扫描行数的关键因素，这个值越小越好。

刚刚我们说了 N 越大，分段数 K 越大。那么， N 固定的时候，什么参数会影响 K 的大小呢？（也就是 λ 的大小）答案是`join_buffer_size`。`join_buffer_size`越大，一次可以放入的行越多，分成的段数也就越少，对被驱动表的全表扫描次数就越少。

这就是为什么，你可能会看到一些建议告诉你，如果你的`join`语句很慢，就把`join_buffer_size`改大。

理解了MySQL执行`join`的两种算法，现在我们再来试着回答文章开头的两个问题。

第一个问题是：能不能使用`join`语句？

1. 如果可以使用**Index Nested-Loop Join**算法，也就是说可以用上被驱动表上的索引，其实是没有问题的；
2. 如果使用**Block Nested-Loop Join**算法，扫描行数就会过多。尤其是在大表上的`join`操作，这样可能要扫描被驱动表很多次，会占用大量的系统资源。所以这种`join`尽量不要用。

所以你在判断要不要使用`join`语句时，就是看`explain`结果里面，`Extra`字段里面有没有出现“**Block Nested Loop**”字样。

第二个问题是：如果要使用`join`，应该选择大表做驱动表还是选择小表做驱动表？

1. 如果是**Index Nested-Loop Join**算法，应该选择小表做驱动表；
2. 如果是**Block Nested-Loop Join**算法：
 - 在`join_buffer_size`足够大的时候，是一样的；
 - 在`join_buffer_size`不够大的时候（这种情况更常见），应该选择小表做驱动表。

所以，这个问题的结论就是，总是应该使用小表做驱动表。

当然了，这里我需要说明下，什么叫作“小表”。

我们前面的例子是没有加条件的。如果我在语句的`where`条件加上`t2.id<=50`这个限定条件，再来看下这两条语句：

```
select * from t1 straight_join t2 on (t1.b=t2.b) where t2.id<=50;  
select * from t2 straight_join t1 on (t1.b=t2.b) where t2.id<=50;
```

注意，为了让两条语句的被驱动表都用不上索引，所以join字段都使用了没有索引的字段**b**。

但如果是用第二个语句的话，**join_buffer**只需要放入**t2**的前50行，显然是更好的。所以这里，“**t2**的前50行”是那个相对小的表，也就是“小表”。

我们再来看另外一组例子：

```
select t1.b,t2.* from t1 straight_join t2 on (t1.b=t2.b) where t2.id<=100;  
select t1.b,t2.* from t2 straight_join t1 on (t1.b=t2.b) where t2.id<=100;
```

这个例子里，表**t1**和**t2**都是只有100行参加join。但是，这两条语句每次查询放入**join_buffer**中的数据是不一样的：

- 表**t1**只查字段**b**，因此如果把**t1**放到**join_buffer**中，则**join_buffer**中只需要放入**b**的值；
- 表**t2**需要查所有的字段，因此如果把表**t2**放到**join_buffer**中的话，就需要放入三个字段**id**、**a**和**b**。

这里，我们应该选择表**t1**作为驱动表。也就是说在这个例子里，“只需要一列参与join的表**t1**”是那个相对小的表。

所以，更准确地说，在决定哪个表做驱动表的时候，应该是两个表按照各自的条件过滤，过滤完成之后，计算参与join的各个字段的总数据量，数据量小的那个表，就是“小表”，应该作为驱动表。

小结

今天，我和你介绍了MySQL执行join语句的两种可能算法，这两种算法是由能否使用被驱动表的索引决定的。而能否用上被驱动表的索引，对join语句的性能影响很大。

通过对Index Nested-Loop Join和Block Nested-Loop Join两个算法执行过程的分析，我们也得到了文章开头两个问题的答案：

1. 如果可以使用被驱动表的索引，join语句还是有其优势的；
2. 不能使用被驱动表的索引，只能使用Block Nested-Loop Join算法，这样的语句就尽量不要使用；
3. 在使用join的时候，应该让小表做驱动表。

最后，又到了今天的问题时间。

我们在上文说到，使用Block Nested-Loop Join算法，可能会因为**join_buffer**不够大，需要对被

驱动表做多次全表扫描。

我的问题是，如果被驱动表是一个大表，并且是一个冷数据表，除了查询过程中可能会导致IO压力大以外，你觉得对这个MySQL服务还有什么更严重的影响吗？（这个问题需要结合上一篇文章的知识点）

你可以把你的结论和分析写在留言区，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章最后留下的问题是，如果客户端由于压力过大，迟迟不能接收数据，会对服务端造成什么严重的影响。

这个问题的核心是，造成了“长事务”。

至于长事务的影响，就要结合我们前面文章中提到的锁、MVCC的知识点了。

- 如果前面的语句有更新，意味着它们在占用着行锁，会导致别的语句更新被锁住；
- 当然读的事务也有问题，就是会导致undo log不能被回收，导致回滚段空间膨胀。

评论区留言点赞板：

@老杨同志 提到了更新之间会互相等锁的问题。同一个事务，更新之后要尽快提交，不要做没必要的查询，尤其是不要执行需要返回大量数据的查询；

@长杰 同学提到了undo表空间变大，db服务堵塞，服务端磁盘空间不足的例子。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Ding Qi, a man with short dark hair and glasses, wearing a black button-down shirt, standing with his arms crossed. To the left of the portrait, the course title "MySQL 实战 45 讲" is displayed in large, bold, dark font. Below the title, a subtitle reads "从原理到实战，丁奇带你搞懂 MySQL". In the bottom left corner, the author's name "林晓斌" is written in a large, stylized font, with smaller text below it identifying him as "网名丁奇" and "前阿里资深技术专家". The overall background is a light gray gradient.

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

精选留言



没时间了ngu

0

join这种用的多的，看完还是有很大收获的。像之前讲的锁之类，感觉好抽象，老是记不住，唉。

2019-01-30

| 作者回复

嗯嗯，因为其实每个同学的只是背景不一样。

这45讲里，每个同学都能从部分文章感觉到有收获，我觉得也很好了！

不过 锁其实用得也多的。。

我以前负责业务库的时候，被开发同学问最多的问题之一就是，为啥死锁了^_^

2019-01-30



抽离の』

8

早上听老师一节课感觉获益匪浅

2019-01-30

| 作者回复

好早呀！

2019-01-30



信信

6

老师好，回答本期问题：如果驱动表分段，那么被驱动表就被多次读，而被驱动表又是大表，循环读取的间隔肯定得超1秒，这就会导致上篇文章提到的：“数据页在LRU_old的存在时间超过1秒，就会移到young区”。最终结果就是把大部分热点数据都淘汰了，导致“Buffer pool hit rate”命中率极低，其他请求需要读磁盘，因此系统响应变慢，大部分请求阻塞。

2019-01-30

| 作者回复

』

2019-01-30



老杨同志

3

对被驱动表进行全表扫描，会把冷数据的page加入到buffer pool，并且block nested-loop要扫描多次，两次扫描的时间可能会超过1秒，使lru的那个优化失效，把热点数据从buffer pool中淘汰掉，影响正常业务的查询效率

2019-01-30

| 作者回复

漂亮！

2019-01-30





壮火虫

1 0

年底了有一种想跳槽的冲动 身在武汉的我想出去看看 可一想到自身的能力和学历 又不敢去了
苦恼...

2019-01-30

作者回复

今年这情况还是要先克制一下^_^

先把内功练起来！

2019-01-30



清风浊酒

1 2

老师您好，`left join` 和 `right join` 会固定驱动表吗？

2019-01-30

作者回复

不会强制，但是由于语义的关系，大概率上是按照语句上写的关系去驱动，效率是比较高的

2019-01-30



柚子

1 2

`join`在热点表操作中，`join`查询是一次给两张表同时加锁吧，会不会增大锁冲突的几率？

业务中肯定要使用被驱动表的索引，通常我们是先在驱动表查出结果集，然后再通过`in`被驱动表索引字段，分两步查询，这样是否比直接`join`委托点？

2019-01-30

作者回复

`join`也是普通查询，都不需要加锁哦，参考下MVCC那篇；

就是我们文中说的，“分两步查询，先查驱动表，然后查多个`in`”，如果可以用上被驱动表的索引，我觉得可以用上Index Nested-Loop Join算法，其实效果是跟拆开写类似的

2019-01-30



郝攀刚

1 1

业务逻辑关系，一个SQL中`left join`7，8个表。这我该怎么优化。每次看到这些脑壳就大！

2019-01-30

作者回复

[]

`Explain`下，没用用`index nested-loop`的全要优化

2019-01-31



Zzz

1 1

林老师，我没想清楚为什么会进入`young`区域。假设大表`t`大小是 M 页>`old`区域 N 页，由于`Block Nested-Loop Join`需要对`t`进行 k 次全表扫描。第一次扫描时， $1 \sim N$ 页依次被放入`old`区域，访问 $N+1$ 页时淘汰1页，放入 $N+1$ 页，以此类推，第一次扫描结束后`old`区域存放的是 $M-N+1 \sim M$ 页。第二次扫描开始，访问1页，淘汰 $M-N+1$ 页，放入1页。可以把 M 页想象成一个环， N 页想象成在这个环上滑动的窗口，由于 $M > N$ ，不管是哪次扫描，需要访问的页都不会在滑动窗口上，所以不会存在“被访问的时候数据页在 LRU 链表中存在的时间超过了 1 秒”而被放入`young`的情况

。我能想到的会被放入young区域的情况是，在当次扫描中，由于一页上有~~多~~行数据，需要对该页访问多次，超过了1s，不管这种情况就和t大小没关系了，而是由于page size太大，而一行数据太少。

2019-01-30

作者回复

你说得对，分两类情况，

小于bp 3/8的情况会跑到young，

大于3/8的会影响young部分的更新

2019-01-30



700

1

老师，您好。看完文章后有如下问题请教：

1) 文章内容「可以看到，在这个查询过程，也是扫描了 200 行，但是总共执行了 101 条语句，比直接 join 多了 100 次交互。除此之外，客户端还要自己拼接 SQL 语句和结果。」这个有没有啥方法来仅通过1次交互就将这101条语句发到服务端执行？

2) 文章内容「每次搜索一棵树近似复杂度是以 2 为底的 M 的对数，记为 $\log_2 M$ ，所以在被驱动表上查一行的时间复杂度是 $2 * \log_2 M$ 。」

这个复杂度的计算难理解，为什么是这么计算？

假设 $M = 256$ ，则搜索树的复杂度为8？

3) 文章内容「因此整个执行过程，近似复杂度是 $N + N * 2 * \log_2 M$ 。」

驱动表的复杂度直接记为 N？

4) 文中提到索引扫描需扫1行数据，全表扫描需扫1000行数据。这是由统计信息决定的？

提前感谢老师！

2019-01-30

作者回复

1. 用 in，但是不建议语句太长

2. 看一下前面我们介绍索引的文章哈

3. 因为是在叶子索引上直接顺序扫描，是一个大致值哈

4. 不是呀，因为表t2是1000行哦

2019-01-30



Ryoma

1

前提：冷数据表 & 大表

buffer pool 中的old区会被持续刷新，并且基本没有升级到young区的可能性。

一定程度上会降低hit rate

2019-01-30



403

0

用那个作为驱动表，mysql会自己优化么？

2019-02-09

| 作者回复

会的

2019-02-10



陈华应

0

老师，放完88行就满了，88是怎么计算得来的呢？

2019-02-02

| 作者回复

这个是实际跑出来的效果

如果说计算的话，每一行固定长度，你用1024除一下

2019-02-02



库淘淘

0

```
set optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
create index idx_c on t2(c);
create index idx_a_c on t1(a,c);
create index idx_b_c on t3(b,c);
mysql> explain select * from t2
-> straight_join t1 on(t1.a=t2.a)
-> straight_join t3 on(t2.b=t3.b)
-> where t1.c> 800 and t2.c>=600 and t3.c>=500;
+-----+-----+-----+-----+-----+-----+-----+
+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | NULL | range | idx_c | idx_c | 5 | NULL | 401 | 100.00 | Using index condition
; Using where; Using MRR |
| 1 | SIMPLE | t1 | NULL | ref | idx_a_c | idx_a_c | 5 | test.t2.a | 1 | 33.33 | Using index condition;
Using join buffer (Batched Key Access) |
| 1 | SIMPLE | t3 | NULL | ref | idx_b_c | idx_b_c | 5 | test.t2.b | 1 | 33.33 | Using index condition;
Using join buffer (Batched Key Access) |
+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

3 rows in set, 1 warning (0.00 sec)

以自己理解考虑如下，有问题请老师能够指出

- 1.根据查询因是select * 肯定回表的，其中在表t2创建索引idx_c,为了能够使用ICP,MRR，如果c字段重复率高或取值行数多，可以考虑不建索引
- 2.已t2 作为驱动表，一方面考虑其他两表都有关联,t2表放入join buffer后关联t1后，再关联t2 得

出结果 再各回t2,t3表取出 得到结果集（之前理解都是t1和t2join得结果集再与t3join，本次理解太确定）

3.t2、t3表建立联合查询目的能够使用ICP

2019-02-01

作者回复

“2.已t2作为驱动表，一方面考虑其他两表都有关联,t2表放入join buffer后关联t1后，再关联t2得出结果 再各回t2,t3表取出 得到结果集”

即使是用t1做驱动表，也是可能可以都用上BKA的哈

新春快乐~

2019-02-04



郭健

0

老师，太棒了！！终于讲join了！！！作为一个实际开发人员，索引了解是必须得，单表索引有所掌握，始终对join没法理解，这节课对我的帮助是最大的。谢谢老师

2019-02-01

作者回复

0

2019-02-01



辣椒

0

我是开发，但是看了老师的专栏，对怎么写数据库应用更有心得了

2019-01-31

作者回复

0，如果有有趣的经验也放到这里跟大家分享哦

2019-02-01



泡泡爱dota

0

explain select * from t1 straight_join t2 on (t1.a=t2.a) where t1.a < 50;

老师，这条sql为什么t1.a的索引没有用上，t1还是走全表

2019-01-31

作者回复

如果数据量不够多，并且满足a<50的行，占比比较高的话，优化器有可能会认为“还要回表，还不如直接扫主键id”

2019-01-31



剃刀吗啡

0

我们某个业务使用infobright这种列式存储，字段没用索引。我在想这种引擎在join的时候是否也会遵守类似的规则？但列式存储并不是按行扫描，所以有点困惑。

2019-01-31

作者回复

是的，只是获取数据的时候，不会去读整行。

但是没有索引就也只能用BNL，可以explain看看

2019-01-31



一大只

0

老师，我想问下，如果使用的是Index Nested-Loop Join，是不是就不会使用join_buffer了？直接将循环结果放到net_buffer_length中，边读边发哈？

2019-01-31

| 作者回复

是的，Index Nested-Loop Join没有用到join buffer

不过35篇马上会介绍到一个优化，把join buffer用上，晚上关注下哦

2019-01-31



斜面镜子 Bill

0

因为 join_buffer 不够大，需要对被驱动表做多次全表扫描，也就造成了“长事务”。除了老师上节课提到的导致undo log 不能被回收，导致回滚段空间膨胀问题，还会出现：1. 长期占用DML锁，引发DDL拿不到锁堵慢连接池；2. SQL执行socket_timeout超时后业务接口重复发起，导致实例IO负载上升出现雪崩；3. 实例异常后，DBA kill SQL因繁杂的回滚执行时间过长，不能快速恢复可用；4. 如果业务采用select *作为结果集返回，极大可能出现网络拥堵，整体拖慢服务端的处理；5. 冷数据污染buffer pool，block nested-loop多次扫描，其中间隔很有可能超过1s，从而污染到lru头部，影响整体的查询体验。

2019-01-31

| 作者回复

很赞

之前知识点的也都加进来啦

2019-01-31

35 | join语句怎么优化?

2019-02-01 林晓斌



在上一篇文章中，我和你介绍了join语句的两种算法，分别是**Index Nested-Loop Join(NLJ)**和**Block Nested-Loop Join(BNL)**。

我们发现在使用**NLJ**算法的时候，其实效果还是不错的，比通过应用层拆分成多个语句然后再拼接查询结果更方便，而且性能也不会差。

但是，**BNL**算法在大表join的时候性能就差多了，比较次数等于两个表参与join的行数的乘积，很消耗**CPU**资源。

当然了，这两个算法都还有继续优化的空间，我们今天就来聊聊这个话题。

为了便于分析，我还是创建两个表t1、t2来和你展开今天的问题。

```
create table t1(id int primary key, a int, b int, index(a));
create table t2 like t1;
drop procedure idata;
delimiter ;;
create procedure idata()
begin
    declare i int;
    set i=1;
    while(i<=1000)do
        insert into t1 values(i, 1001-i, i);
        set i=i+1;
    end while;

    set i=1;
    while(i<=1000000)do
        insert into t2 values(i, i, i);
        set i=i+1;
    end while;

end;;
delimiter ;
call idata();
```

为了便于后面量化说明，我在表t1里，插入了1000行数据，每一行的a=1001-id的值。也就是说，表t1中字段a是逆序的。同时，我在表t2中插入了100万行数据。

Multi-Range Read优化

在介绍join语句的优化方案之前，我需要先和你介绍一个知识点，即： Multi-Range Read优化(MRR)。这个优化的主要目的是尽量使用顺序读盘。

在[第4篇文章](#)中，我和你介绍InnoDB的索引结构时，提到了“回表”的概念。我们先来回顾一下这个概念。回表是指， InnoDB在普通索引a上查到主键id的值后，再根据一个个主键id的值到主键索引上去查整行数据的过程。

然后，有同学在留言区问到，回表过程是一行行地查数据，还是批量地查数据？

我们先来看看这个问题。假设，我执行这个语句：

```
select * from t1 where a>=1 and a<=100;
```

主键索引是一棵B+树，在这棵树上，每次只能根据一个主键id查到一行数据。因此，回表肯定是一行行搜索主键索引的，基本流程如图1所示。

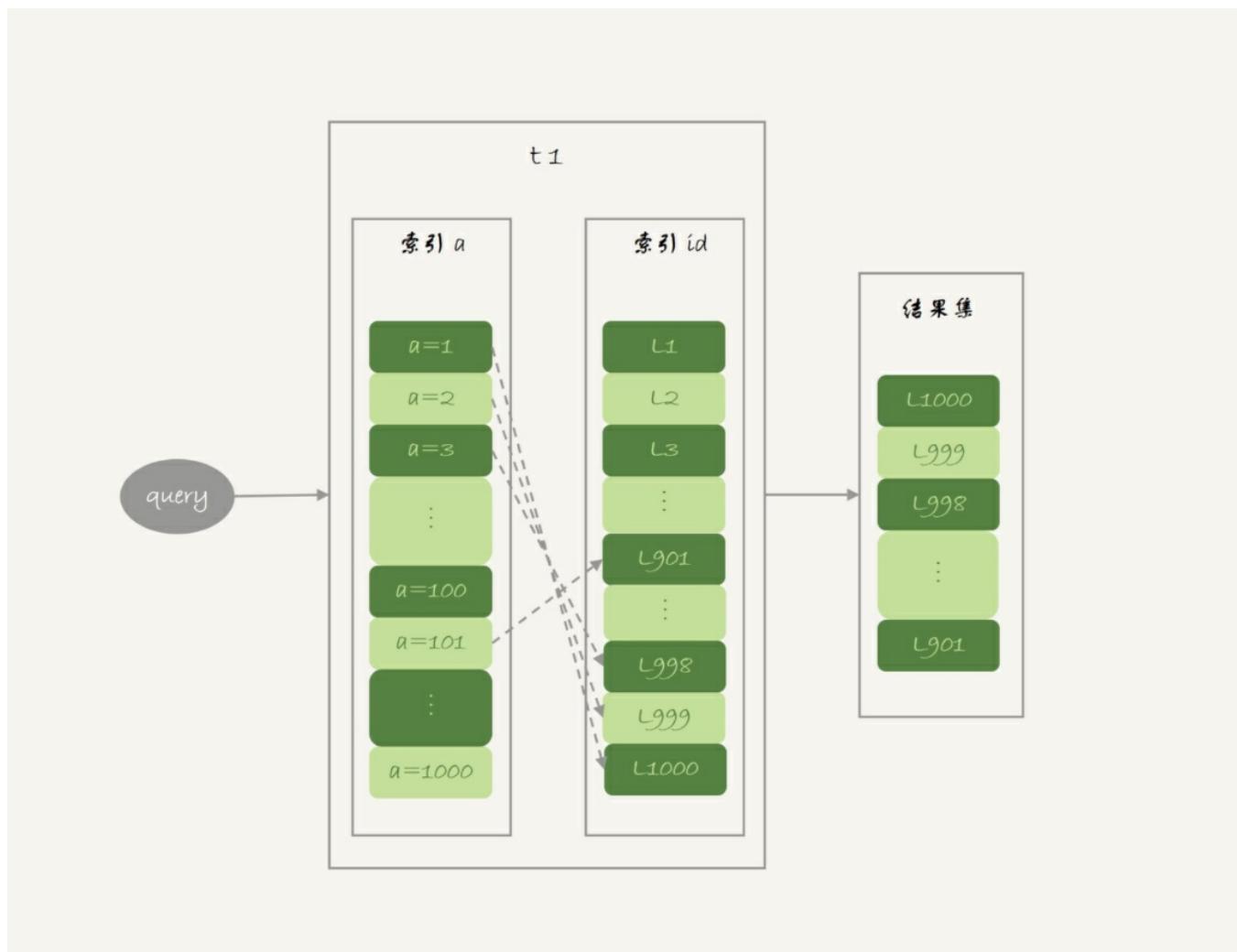


图1 基本回表流程

如果随着a的值递增顺序查询的话，id的值就变成随机的，那么就会出现随机访问，性能相对较差。虽然“按行查”这个机制不能改，但是调整查询的顺序，还是能够加速的。

因为大多数的数据都是按照主键递增顺序插入得到的，所以我们可以认为，如果按照主键的递增顺序查询的话，对磁盘的读比较接近顺序读，能够提升读性能。

这，就是MRR优化的设计思路。此时，语句的执行流程变成了这样：

1. 根据索引a，定位到满足条件的记录，将id值放入read_md_buffer中；
2. 将read_md_buffer中的id进行递增排序；
3. 排序后的id数组，依次到主键id索引中查记录，并作为结果返回。

这里，`read_rnd_buffer`的大小是由`read_rnd_buffer_size`参数控制的。如果步骤1中，`read_rnd_buffer`放满了，就会先执行完步骤2和3，然后清空`read_rnd_buffer`。之后继续找索引a的下个记录，并继续循环。

另外需要说明的是，如果你想要稳定地使用MRR优化的话，需要设置`set optimizer_switch="mrr_cost_based=off"`。（官方文档的说法，是现在的优化器策略，判断消耗的时候，会更倾向于不使用MRR，把`mrr_cost_based`设置为`off`，就是固定使用MRR了。）

下面两幅图就是使用了MRR优化后的执行流程和explain结果。

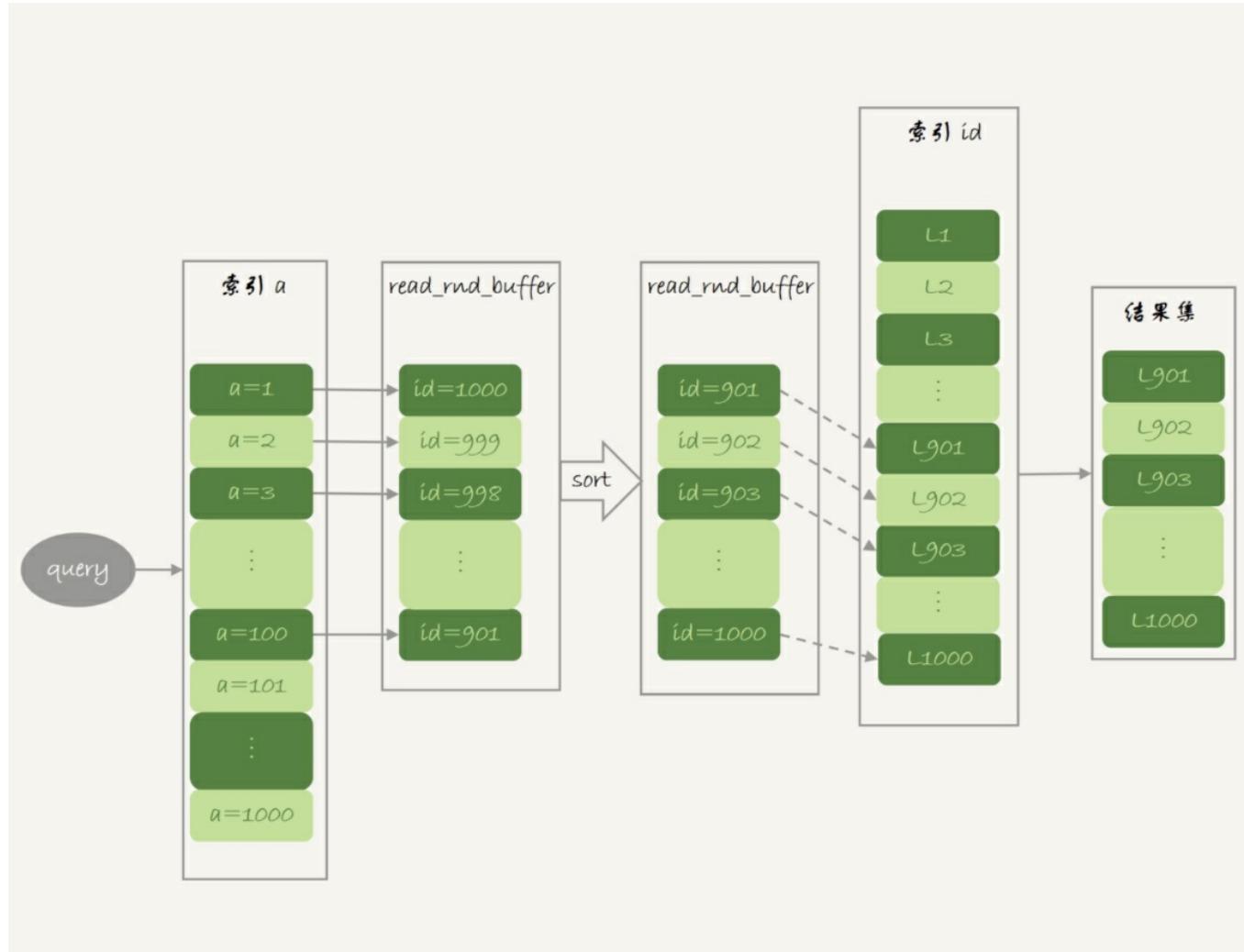


图2 MRR执行流程

mysql> explain select * from t2 where a>=100 and a<=200;											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t2	NULL	range	a	a	5	NULL	101	100.00	Using index condition; Using MRR

图3 MRR执行流程的explain结果

从图3的explain结果中，我们可以看到`Extra`字段多了`Using MRR`，表示的是用上了MRR优化。而且，由于我们在`read_rnd_buffer`中按照`id`做了排序，所以最后得到的结果集也是按照主键`id`递增顺序的，也就是与图1结果集中行的顺序相反。

到这里，我们小结一下。

MRR能够提升性能的核心在于，这条查询语句在索引[a](#)上做的是一个范围查询（也就是说，这是一个多值查询），可以得到足够多的主键id。这样通过排序以后，再去主键索引查数据，才能体现出“顺序性”的优势。

Batched Key Access

理解了**MRR**性能提升的原理，我们就能理解**MySQL**在5.6版本后开始引入的**Batched Key Access(BKA)**算法了。这个**BKA**算法，其实就是对**NLJ**算法的优化。

我们再来看看上一篇文章中用到的**NLJ**算法的流程图：

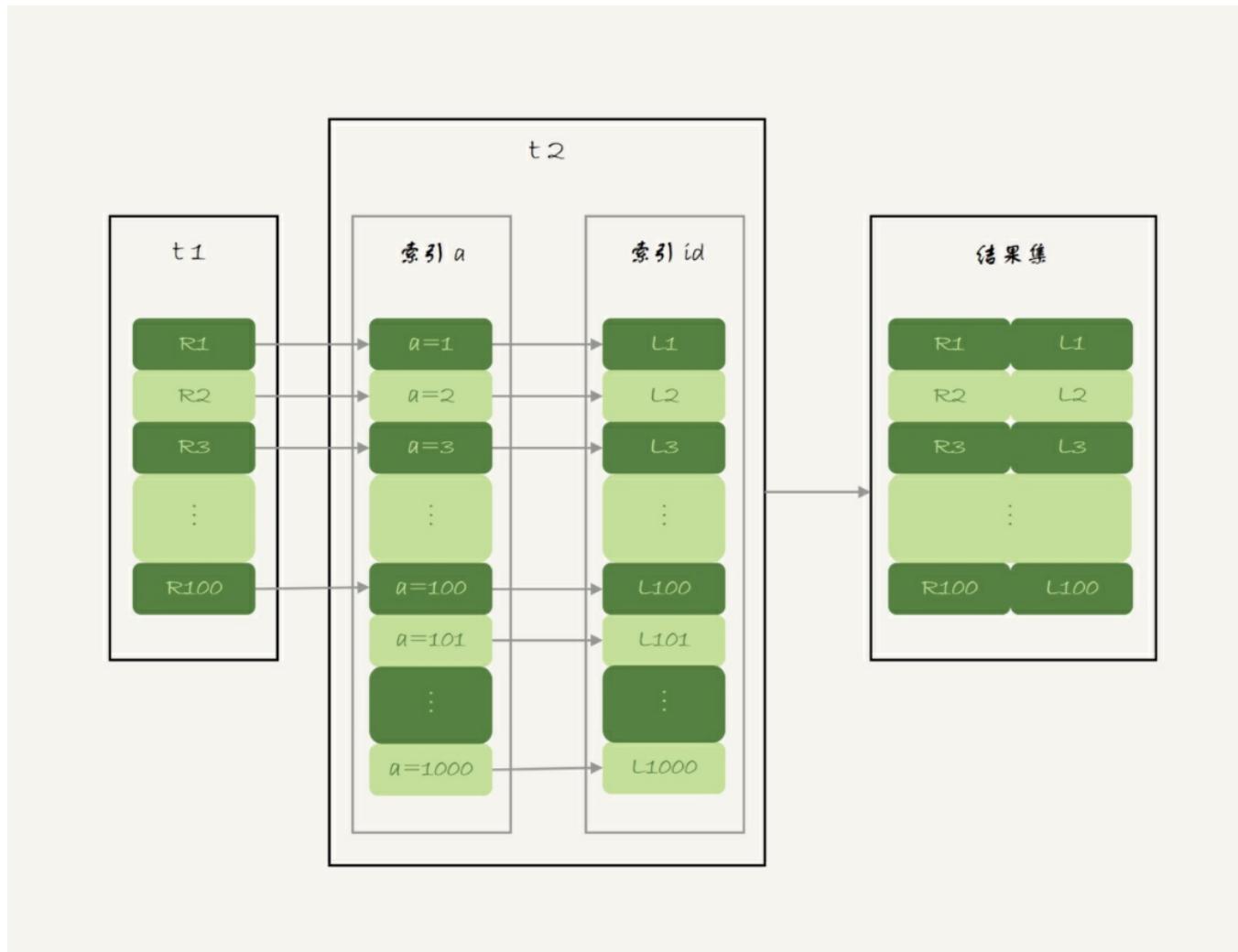


图4 Index Nested-Loop Join流程图

NLJ算法执行的逻辑是：从驱动表t1，一行行地取出a的值，再到被驱动表t2去做join。也就是说，对于表t2来说，每次都是匹配一个值。这时，**MRR**的优势就用不上了。

那怎么才能一次性地多传些值给表t2呢？方法就是，从表t1里一次性地多拿些行出来，一起传给表t2。

既然如此，我们就把表t1的数据取出来一部分，先放到一个临时内存。这个临时内存不是别人，

就是join_buffer。

通过上一篇文章，我们知道join_buffer在BNL算法里的作用，是暂存驱动表的数据。但是在NLJ算法里并没有用。那么，我们刚好就可以复用join_buffer到BKA算法中。

如图5所示，是上面的NLJ算法优化后的BKA算法的流程。

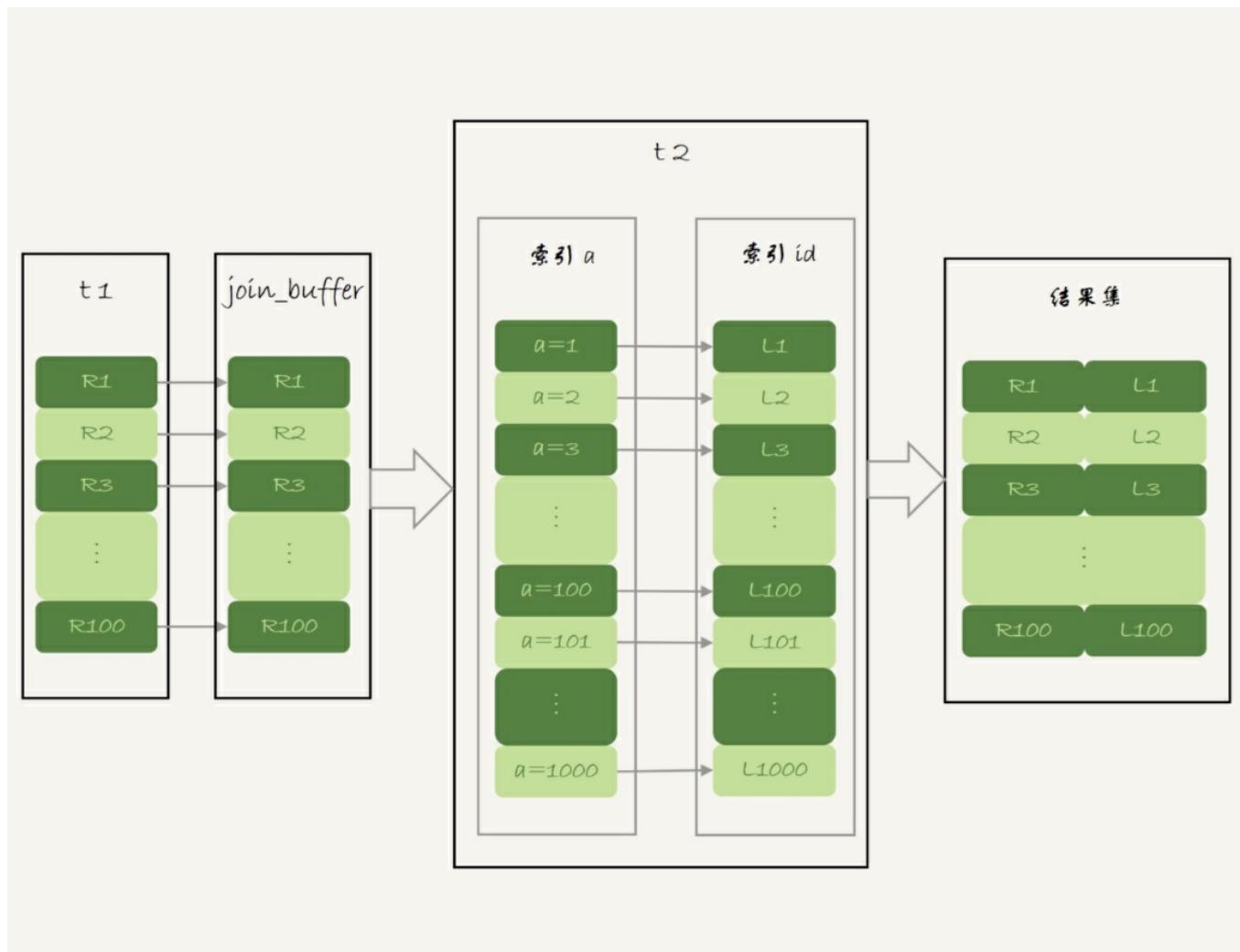


图5 Batched Key Access流程

图中，我在join_buffer中放入的数据是P1~P100，表示的是只会取查询需要的字段。当然，如果join buffer放不下P1~P100的所有数据，就会把这100行数据分成多段执行上图的流程。

那么，这个BKA算法到底要怎么启用呢？

如果要使用BKA优化算法的话，你需要在执行SQL语句之前，先设置

```
set optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

其中，前两个参数的作用是要启用MRR。这么做的原因是，BKA算法的优化要依赖于MRR。

BNL算法的性能问题

说完了NLJ算法的优化，我们再来看BNL算法的优化。

我在上一篇文章末尾，给你留下的思考题是，使用**Block Nested-Loop Join(BNL)**算法时，可能会对被驱动表做多次扫描。如果这个被驱动表是一个大的冷数据表，除了会导致**IO**压力大以外，还会对系统有什么影响呢？

在[第33篇文章](#)中，我们说到InnoDB的LRU算法的时候提到，由于InnoDB对Buffer Pool的LRU算法做了优化，即：第一次从磁盘读入内存的数据页，会先放在old区域。如果1秒之后这个数据页不再被访问了，就不会被移动到LRU链表头部，这样对Buffer Pool的命中率影响就不大。

但是，如果一个使用BNL算法的join语句，多次扫描一个冷表，而且这个语句执行时间超过1秒，就会在再次扫描冷表的时候，把冷表的数据页移到LRU链表头部。

这种情况对应的，是冷表的数据量小于整个Buffer Pool的3/8，能够完全放入old区域的情况。

如果这个冷表很大，就会出现另外一种情况：业务正常访问的数据页，没有机会进入young区域。

由于优化机制的存在，一个正常访问的数据页，要进入young区域，需要隔1秒后再次被访问到。但是，由于我们的join语句在循环读磁盘和淘汰内存页，进入old区域的数据页，很可能在1秒之内就被淘汰了。这样，就会导致这个MySQL实例的Buffer Pool在这段时间内，young区域的数据页没有被合理地淘汰。

也就是说，这两种情况都会影响Buffer Pool的正常运作。

大表join操作虽然对IO有影响，但是在语句执行结束后，对IO的影响也就结束了。但是，对Buffer Pool的影响就是持续性的，需要依靠后续的查询请求慢慢恢复内存命中率。

为了减少这种影响，你可以考虑增大join_buffer_size的值，减少对被驱动表的扫描次数。

也就是说，BNL算法对系统的影响主要包括三个方面：

1. 可能会多次扫描被驱动表，占用磁盘IO资源；
2. 判断join条件需要执行M*N次对比（M、N分别是两张表的行数），如果是大表就会占用非常多的CPU资源；
3. 可能会导致Buffer Pool的热数据被淘汰，影响内存命中率。

我们执行语句之前，需要通过理论分析和查看explain结果的方式，确认是否要使用BNL算法。如果确认优化器会使用BNL算法，就需要做优化。优化的常见做法是，给被驱动表的join字段加上索引，把BNL算法转成BKA算法。

接下来，我们就具体看看，这个优化怎么做？

BNL转BKA

一些情况下，我们可以直接在被驱动表上建索引，这时就可以直接转成BKA算法了。

但是，有时候你确实会碰到一些不适合在被驱动表上建索引的情况。比如下面这个语句：

```
select * from t1 join t2 on (t1.b=t2.b) where t2.b>=1 and t2.b<=2000;
```

我们在文章开始的时候，在表t2中插入了100万行数据，但是经过where条件过滤后，需要参与join的只有2000行数据。如果这条语句同时是一个低频的SQL语句，那么再为这个语句在表t2的字段b上创建一个索引就很浪费了。

但是，如果使用BNL算法来join的话，这个语句的执行流程是这样的：

1. 把表t1的所有字段取出来，存入join_buffer中。这个表只有1000行，join_buffer_size默认值是256k，可以完全存入。
2. 扫描表t2，取出每一行数据跟join_buffer中的数据进行对比，
 - 如果不满足t1.b=t2.b，则跳过；
 - 如果满足t1.b=t2.b，再判断其他条件，也就是是否满足t2.b处于[1,2000]的条件，如果是，就作为结果集的一部分返回，否则跳过。

我在上一篇文章中说过，对于表t2的每一行，判断join是否满足的时候，都需要遍历join_buffer中的所有行。因此判断等值条件的次数是1000*100万=10亿次，这个判断的工作量很大。

mysql> explain select * from t1 join t2 on (t1.b=t2.b) where t2.b>=1 and t2.b<=2000;											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t1	NULL	ALL	NULL	NULL	NULL	NULL	1000	100.00	Using where
1	SIMPLE	t2	NULL	ALL	NULL	NULL	NULL	NULL	998222	1.11	Using where; Using join buffer (Block Nested Loop)

图6 explain结果

998	3	998	998	998	998	998
999	2	999	999	999	999	999
1000	1	1000	1000	1000	1000	1000
+-----+						
1000 rows in set (1 min 11.95 sec)						

图7 语句执行时间

可以看到，explain结果里Extra字段显示使用了BNL算法。在我的测试环境里，这条语句需要执

行1分11秒。

在表t2的字段b上创建索引会浪费资源，但是不创建索引的话这个语句的等值条件要判断10亿次，想想也是浪费。那么，有没有两全其美的办法呢？

这时候，我们可以考虑使用临时表。使用临时表的大致思路是：

1. 把表t2中满足条件的数据放在临时表tmp_t中；
2. 为了让join使用BKA算法，给临时表tmp_t的字段b加上索引；
3. 让表t1和tmp_t做join操作。

此时，对应的SQL语句的写法如下：

```
create temporary table temp_t(id int primary key, a int, b int, index(b))engine=innodb;
insert into temp_t select * from t2 where b>=1 and b<=2000;
select * from t1 join temp_t on (t1.b=temp_t.b);
```

图8就是这个语句序列的执行效果。

```
mysql> create temporary table temp_t(id int primary key, a int, b int, index(b))engine=innodb;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into temp_t select * from t2 where b>=1 and b<=2000;
Query OK, 2000 rows affected (0.90 sec)
Records: 2000  Duplicates: 0  Warnings: 0

mysql> explain select * from t1 join temp_t on (t1.b=temp_t.b);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | partitions | type | possible_keys | key | key_len | ref   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE      | t1   | NULL       | ALL  | NULL          | NULL | NULL    | NULL   |
| 1   | SIMPLE      | temp_t | NULL       | ref  | b             | b   | 5       | test.t1.b |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
996 | 5 | 996 | 996 | 996 | 996 |
997 | 4 | 997 | 997 | 997 | 997 |
998 | 3 | 998 | 998 | 998 | 998 |
999 | 2 | 999 | 999 | 999 | 999 |
1000 | 1 | 1000 | 1000 | 1000 | 1000 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1000 rows in set (0.01 sec)
```

图8 使用临时表的执行效果

可以看到，整个过程3个语句执行时间的总和还不到1秒，相比于前面的1分11秒，性能得到了大幅提升。接下来，我们一起看一下这个过程的消耗：

1. 执行insert语句构造temp_t表并插入数据的过程中，对表t2做了全表扫描，这里扫描行数是100万。

- 之后的join语句，扫描表t1，这里的扫描行数是1000；join比较过程中，做了1000次带索引的查询。相比于优化前的join语句需要做10亿次条件判断来说，这个优化效果还是很明显的。

总体来看，不论是在原表上加索引，还是用有索引的临时表，我们的思路都是让join语句能够用上被驱动表上的索引，来触发BKA算法，提升查询性能。

扩展-hash join

看到这里你可能发现了，其实上面计算10亿次那个操作，看上去有点儿傻。如果join_buffer里面维护的不是一个无序数组，而是一个哈希表的话，那么就不是10亿次判断，而是100万次hash查找。这样的话，整条语句的执行速度就快多了吧？

确实如此。

这，也正是MySQL的优化器和执行器一直被诟病的一个原因：不支持哈希join。并且，MySQL官方的roadmap，也是迟迟没有把这个优化排上议程。

实际上，这个优化思路，我们可以自己实现在业务端。实现流程大致如下：

- select * from t1;取得表t1的全部1000行数据，在业务端存入一个hash结构，比如C++里的set、PHP的dict这样的数据结构。
- select * from t2 where b>=1 and b<=2000; 获取表t2中满足条件的2000行数据。
- 把这2000行数据，一行一行地取到业务端，到hash结构的数据表中寻找匹配的数据。满足匹配的条件的这行数据，就作为结果集的一行。

理论上，这个过程会比临时表方案的执行速度还要快一些。如果你感兴趣的话，可以自己验证一下。

小结

今天，我和你分享了Index Nested-Loop Join (NLJ) 和 Block Nested-Loop Join (BNL) 的优化方法。

在这些优化方法中：

- BKA优化是MySQL已经内置支持的，建议你默认使用；
- BNL算法效率低，建议你都尽量转成BKA算法。优化的方向就是给被驱动表的关联字段加上索引；
- 基于临时表的改进方案，对于能够提前过滤出小数据的join语句来说，效果还是很好的；

4. MySQL目前的版本还不支持**hash join**, 但你可以配合应用端自己模拟出来, 理论上效果要好于临时表的方案。

最后, 我给你留下一道思考题吧。

我们在讲**join**语句的这两篇文章中, 都只涉及到了两个表的**join**。那么, 现在有一个三个表**join**的需求, 假设这三个表的表结构如下:

```
CREATE TABLE `t1` (
  `id` int(11) NOT NULL,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

```
create table t2 like t1;
create table t3 like t2;
insert into ... //初始化三张表的数据
```

语句的需求实现如下的**join**逻辑:

```
select * from t1 join t2 on(t1.a=t2.a) join t3 on (t2.b=t3.b) where t1.c>=X and t2.c>=Y and t3.c>=Z;
```

现在为了得到最快的执行速度, 如果让你来设计表**t1**、**t2**、**t3**上的索引, 来支持这个**join**语句, 你会加哪些索引呢?

同时, 如果我希望你用**straight_join**来重写这个语句, 配合你创建的索引, 你就需要安排连接顺序, 你主要考虑的因素是什么呢?

你可以把你的方案和分析写在留言区, 我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听, 也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上篇文章最后留给你的问题, 已经在本篇文章中解答了。

这里我再根据评论区留言的情况, 简单总结下。根据数据量的大小, 有这么两种情况:

- @长杰 和 @老杨同志 提到了数据量小于**old**区域内存的情况;

- @Zzz 同学，很认真地看了其他同学的评论，并且提了一个很深的问题。对被驱动表数据量大于Buffer Pool的场景，做了很细致的推演和分析。

给这些同学点赞，非常好的思考和讨论。

The image shows the cover of a MySQL practical course. At the top left is the 'Geektime' logo. The title 'MySQL 实战 45讲' is prominently displayed in large, bold, black font. Below it, a subtitle reads '从原理到实战，丁奇带你搞懂 MySQL'. To the right of the text is a portrait of the instructor, Lin Xiaobin (丁奇), wearing glasses and a dark shirt, with his arms crossed. On the left side of the cover, there is a photo of a person's face. Below the title area, the author's name '林晓斌' and nickname '丁奇' are listed, along with the text '前阿里资深技术专家'. At the bottom of the cover, there is a promotional message: '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。' (New version upgraded: Click 'Share with friends', get 10 free reads, and there are cash rewards for subscriptions.)

精选留言



郭健

2

老师，有几个问题还需要请教一下：

1. 上一章 t1 表 100 条数据，t2 1000 条数据，mysql 会每次都会准确的找出哪张表是合理的驱动表吗？还是需要人为的添加 straight_join。
2. 像 left join 这种，左边一定是驱动表吧？以左边为标准查看右边有符合的条件，拼成一条数据，看到你给其他同学的评论说可能不是，这有些疑惑。
3. 在做 join 的时候，有些条件是可以放在 on 中也可以放在 where 中，比如 (t1.yn=1 和 t2.yn=1) 这种简单判断是否删除的。最主要的是，需要根据两个条件才能 join 的 (productCode 和 custCode)，需要两个都在 on 里，还是一个在 on 中，一个在 where 中更好呢？

2019-02-07

| 作者回复

1. 正常是会自己找到合理的，但是用前 explain 是好习惯哈
2. 这个问题的展开我放到答疑文章中哈
3. 这也是好问题，需要分析是使用哪种算法，也放到答疑文章展开哈。

新年快乐~

2019-02-07



Geek_02538c

1

过年了，还有新文章，给个赞。另，**where** 和 **order** 与索引的关系，都讲过了，**group by** 是否也搞个篇章说一下。

2019-02-02

| 作者回复

你说得对^_^ 第37篇就是，新年快乐

2019-02-03



Ryoma

1

`read_rnd_buffer_length` 参数应该是 `read_rnd_buffer_size`, 见文档: https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_read_rnd_buffer_size

2019-02-02

| 作者回复

你说得对，多谢

发起勘误了

新年快乐

2019-02-03



Mr.Strive.Z.H.L

1

老师您好，新年快乐~~

关于三表**join**有一个疑惑点需要确认：

老师您在评论中说到，三表**join**不会是前两个表**join**后得到结果集，再和第三张表**join**。

针对这句话，我的理解是：

假设我们不考虑**BKA**，就按照一行行数据来判断的话，流程应该如下（我会将**server**端和**innodb**端分得很清楚）：

表是t1 ,t2 ,t3。 t1 **straight_join** t2 **straight_join** t3，这样的**join**顺序。

1. 调用**innodb**接口，从t1中取一行数据，数据返回到**server**端。

2. 调用**innodb**接口，从t2中取满足条件的数据，数据返回到**server**端。

3. 调用**innodb**接口，从t3中取满足条件的数据，数据返回到**server**端。

上面三步之后，驱动表 t1的一条数据就处理完了，接下来重复上述过程。

（如果采用**BKA**进行优化，可以理解为不是一行行数据的提取，而是一个范围内数据的提取）

。

按照我上面的描述，确实没有前两表先**join**得结果集，然后再**join**第三张表的过程。

不知道我上面的描述的流程对不对？（我个人觉得，将**innodb**的处理和**server**端的处理分隔清晰，对于**sql**语句的理解，会透彻很多）

2019-02-02

作者回复

新年快乐，分析得很好。

可以再补充一句，会更好理解你说的这个过程：

如果采用BKA进行优化,每多一个join, 就多一个join_buffer

2019-02-02



LY

1

order by cjsj desc limit 0,20 explain Extra只是显示 Using where , 执行时间 7秒钟

order by cjsj desc limit 5000,20 explain Extra只是显示 Using index condition; Using where; Using filesort, 执行时间 0.1 秒

有些许的凌乱了@^^@

2019-02-01

作者回复

这正常的，一种可能是这样的：

Using where 就是顺序扫，但是这个上要扫很久才能扫到满足条件的20个记录；

虽然有filesort，但是如果参与排序的行数少，可能速度就更快，而且limit 有堆排序优化哦

2019-02-01



郭健

0

老师，新年快乐！！看到您给我提问的回答，特别期待您之后的答疑，因为dba怕我们查询数据库时连接时间过长，影响线上实际运行。所以就开发出一个网页，让我们进行查询，但是超过几秒(具体不知道，查询一个200w的数据，条件没有加索引有时候都会)就会返回time out，所以当查询大表并join的时候，就会很吃力！想法设法的缩小单位，一般我们都不会为createTime建一个索引，所以在根据时间缩小范围的时候有时候也并不是很好的选择。我们线上做统计sql的时候，因为数据量比较大，筛选条件也比较多，一个sql可能在0.4s多，这已经是属于慢sql了。感谢老师对我提问的回答！！

2019-02-09



磊

0

一直对多表的join有些迷惑，希望老师后面专门把这块给讲的透彻些

2019-02-03

作者回复

这一期45篇 join差不多就讲这些了！

有问题在评论区提出来哈

2019-02-03



bluefantasy3

0

请教老师一个问题：innodb的Buffer Pool的内存是innodb自己管理还是使用OS的page cache？我理解应该是innodb自己管理。我在另一个课程里看到如果频繁地把OS的/proc/sys/vm/drop_caches 改成 1会影响MySQL的性能，如果buffer pool是MySQL自己管理，应该不受这个参数影

响呀？请解答。

2019-02-02

作者回复

1. 是MySQL自己管理的
2. 一般只有数据文件是o_direct的，redo log和binlog都是有用到文件系统的page cache，因此多少有影响的

好问题

2019-02-03



信信

0

老师好，有两点疑问请老师解惑：

- 1、图8上面提到的关于临时表的第三句是不是还是使用straight_join好一些？不然有可能temp_t被选为驱动表？
- 2、图8下面提到join过程中做了1000次带索引的查询，这里的1000也是在打开mrr的情况下的吗？是进行了1000次树搜索，还是找到第一个后，依次挨着读呢？

2019-02-02

作者回复

1. 写straight_join能确定顺序，也可以的，这里写join也ok的

2. 是进行了1000次树搜索

2019-02-02



HuaMax

0

前提假设：t1.c>=X可以让t1成为小表。同时打开BKA和MRR。

- 1、t1表加(c,a)索引。理由：A、t1.c>=X可以使用索引；B、加上a的联合索引，join buffer里放入的是索引(c,a)而不是去主键表取整行，用于与表t2的t1.a = t2.a的join查询，不过返回SELECT * 最终还是需要回表。
- 2、t2表加(a,b,c)索引。理由：A、加上a避免与t1表join查询的BNL；B、理由同【1-B】；C、加上c不用回表判断t2.c>=Y的筛选条件
- 3、t3表加(b,c)索引。理由：A、避免与t2表join查询的BNL；C、理由同【2-C】

问题：

- 1、【1-B】和【2-B】由于select *要返回所有列数据，不敢肯定join buffer里是回表的整行数据还是索引(c,a)的数据，需要老师解答一下；不过值得警惕的是，返回的数据对sql的执行策略有非常大的影响。
- 2、在有join查询时，被驱动表是先做join连接查询，还是先筛选数据再从筛选后的临时表做join连接？这将影响上述的理由【2-C】和【3-C】

使用straight_join强制指定驱动表，我会改写成这样：select * from t2 STRAIGHT_JOIN t1 on(t1.a=t2.a) STRAIGHT_JOIN t3 on (t2.b=t3.b) where t1.c>=X and t2.c>=Y and t3.c>=Z;

考虑因素包括：

- 1、驱动表使用过滤条件筛选后的数据量，使其成为小表，上面的改写也是基于t2是小表

2、因为t2是跟t1,t3都有关联查询的，这样的话我猜测对t1,t3的查询是不是可以并行执行，而如果使用t1,t3作为主表的话，是否会先跟t2生成中间表，是个串行的过程？

3、需要给t1加(a,c)索引，给t2加(c,a,b)索引。

2019-02-02

| 作者回复

[] 很深入的思考哈

1. select *，所以放整行；你说得对，select * 不是好习惯；

2. 第一次join后就筛选；第二次join再筛选；

新春快乐~

2019-02-04



库淘淘

0

```
set optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
create index idx_c on t2(c);
create index idx_a_c on t1(a,c);
create index idx_b_c on t3(b,c);
mysql> explain select * from t2
-> straight_join t1 on(t1.a=t2.a)
-> straight_join t3 on(t2.b=t3.b)
-> where t1.c> 800 and t2.c>=600 and t3.c>=500;
+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+
| 1 | SIMPLE | t2 | NULL | range | idx_c | idx_c | 5 | NULL | 401 | 100.00 | Using index condition
; Using where; Using MRR |
| 1 | SIMPLE | t1 | NULL | ref | idx_a_c | idx_a_c | 5 | test.t2.a | 1 | 33.33 | Using index conditio
n; Using join buffer (Batched Key Access) |
| 1 | SIMPLE | t3 | NULL | ref | idx_b_c | idx_b_c | 5 | test.t2.b | 1 | 33.33 | Using index conditio
n; Using join buffer (Batched Key Access) |
+-----+-----+-----+-----+
3 rows in set, 1 warning (0.00 sec)
```

以自己理解如下，有问题请老师能够指出

1.根据查询因是select * 肯定回表的，其中在表t2创建索引idx_c,为了能够使用ICP,MRR，如果c字段重复率高或取值行数多，可以考虑不建索引

2.已t2作为驱动表，一方面考虑其他两表都有关联,t2表放入join buffer后关联t1后，再关联t2得出结果再各回t2,t3表取出 得到结果集（之前理解都是t1和t2join得结果集再与t3join，本次理解太确定）

3.t2、t3表建立联合查询目的能够使用ICP

2019-02-01

| 作者回复

[]

BKA是从Index Nested-Loop join 优化而来的，并不是“t1和t2join得结果集再与t3join”，而是直接嵌套循环执行下去。

这个效果相当不错了，MRR, BKA都用上

2019-02-02



LY

0

刚刚凌乱了的那个问题，经explain验证，
explain SELECT a.* FROM sys_xxtx a JOIN baq_ryxx
r ON a.ryid = r.ID WHERE a.dwbh = '7E0A13A14101D0A8E0430A0F23BCD0A8' ORDER BY tx
sj DESC LIMIT 0,20;

使用的索引是txsj；

explain SELECT a.* FROM sys_xxtx a JOIN baq_ryxx r ON a.ryid = r.ID WHERE a.dwbh = '7E0
A13A14101D0A8E0430A0F23BCD0A8' ORDER BY txsj DESC LIMIT 5000,20;使用的索引是dw
bh；

然后回忆起了第10张：MySQL为什么有时候会选错索引？

但是从扫描行数、是否使用排序等来看在 LIMIT 5000,20时候也应该优选txsj？可是这个时候选
择的索引是dwbh，查询时间也大大缩短

2019-02-01

| 作者回复

嗯，这个跟我们第十篇那个例子挺像的

我们把limit 1改成limit 100的时候，MySQL认为，要扫描到“100行那么多”，

你这里是limit 5000, 200，这个5000会让优化器认为，选txsj会要扫“很多行，可能很久”

这个确实是优化器还不够完善的地方，有时候不得不用force index~

2019-02-02



dzkk

0

老师，对于关联查询（inner join），个人有几点理解，请帮助审核是否正确，谢了。

正确选择：

结果集小的为驱动表，且被驱动表有索引

未知效果选择：

1) 结果集小的为驱动表，但是被驱动表没有索引

2) 结果集大的为驱动表，但是被驱动表有索引

最差选择：

结果集大的为驱动表，且被驱动表没有索引

2019-02-01

| 作者回复

未知效果选择 是啥意思^_^

2019-02-02



老杨同志

0

我准备给

t1增加索引c

t2增加组合索引b,c

t3增加组合索引b,c

```
select * from t1 straight_join t2 on(t1.a=t2.a) straight_join t3 on (t2.b=t3.b) where t1.c>=X and t2.c>=Y and t3.c>=Z;
```

另外我还有个问题，开篇提到的这句sql `select * from t1 where a>=1 and a<=100;`

a是索引列，如果这句索引有`order by a`，不使用MRR优化，查询出来就是按a排序的，使用了mrr优化，是不是要额外排序

2019-02-01

| 作者回复

对，好问题，用了`order by`就不用MRR了

2019-02-02



poppy

0

```
select * from t1 join t2 on(t1.a=t2.a) join t3 on (t2.b=t3.b) where t1.c>=X and t2.c>=Y and t3.c>=Z;
```

老师，我的理解是真正做join的三张表的大小实际上是`t1.c>=X`、`t2.c>=Y`、`t3.c>=Z`对应满足条件的行数，为了方便快速定位到满足条件的数据，t1、t2和t3的c字段最好都建索引。对于join操作，按道理mysql应该会优先选择join之后数量比较少的两张表先来进行join操作，例如满足`t1.a=t2.a`的行数小于满足`t2.b=t3.b`的行数，那么就会优先将t1和t2进行join，选择`t1.c>=X`、`t2.c>=Y`中行数少的表作为驱动表，另外一张作为被驱动表，在被驱动表的a的字段上建立索引，这样就完成了t1和t2的join操作并把结果放入join_buffer准备与t3进行join操作，则在作为被驱动表的t3的b字段上建立索引。不知道举的这个例子分析得是否正确，主要是这里不知道t1、t2、t3三张表的数据量，以及满足`t1.c>=X`，`t2.c>=Y`，`t3.c>=Z`的数据量，还有各个字段的区分度如何，是否适合建立索引等。

2019-02-01

| 作者回复

嗯 这个问题就是留给大家自己设定条件然后分析的，分析得不错哦

2019-02-02



Destroy、

0

BNL 算法效率低，建议你都尽量转成 BKA 算法。优化的方向就是给驱动表的关联字段加上索引；

老师最后总结的时候，这句话后面那句，应该是给被驱动表的关联字段加上索引吧。

2019-02-01

| 作者回复

对的，细致

已经发起勘误，谢谢你哦，新年快乐

2019-02-01



LY

0



YEAR(txsj) = '2018' 有结果集, YEAR(txsj) = '2019' 无结果集,
YEAR(txsj) = '2018' 和 YEAR(txsj) = '2019' 查询所需时间 后者是前者的10倍
请老师分析下大概什么原因?

2019-02-01

| 作者回复

这个信息太不足了!

我第一时间反应是不是有limit?

你给贴一下表结构,

sql语句, 还有explain这个语句的结果, 我们再来分析下哈

2019-02-01



John

0

期待這一篇很久啦 終於出來啦 臨時表和範圍搜索實在是醍醐灌頂 謝謝老師

2019-02-01



永恒记忆

0

老师, 记得之前看目录之后要将一篇标题大概为“我的mysql为啥莫名其妙重启了”, 最近看怎么没有了? 我们确实遇到这种问题, 在系统日志里也找不到OOM信息, 现象是半个月左右就会自动重启一下, 时间不固定, 想请教下是什么问题呢?

2019-02-01

| 作者回复

贴一下errorlog里面看看有没有异常信息 如果比较大的文件可以发我微博私信附件

写文章的过程中根据大家的评论问题, 发现有些知识点应该优先写, 目录有做调整哈

2019-02-01



郭江伟

0

select * from t1 join t2 on(t1.a=t2.a) join t3 on (t2.b=t3.b) where t1.c>=X and t2.c>=Y and t3.c>=Z;

这个语句建索引需要考虑三个表的数据量和相关字段的数据分布、选择率、每个条件返回行数占比等

我的测试场景是:

t1 1000行数据 t2 100w行数据 t3 100w行, 关联字段没有重复值, 条件查询返回行数占比很少, 此时索引为:

```
alter table t1 add key t1_c(c);
alter table t2 add key t2_ac(a,c);
alter table t3 add key t3_bc(b,c);
```

测试sql无索引是执行需要2分钟多, 加了索引后需要0.01秒, 加索引后执行计划为:

```
mysql> explain select * from t1 join t2 on(t1.a=t2.a) join t3 on (t2.b=t3.b) where t1.c>=100 and t2.c>=10 and t3.c>=90;
```

```
+-----+-----+-----+-----+-----+-----+-----+
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t1	NULL	ALL	t1_a	NULL	NULL	NULL	1000	90.10	Using where
1	SIMPLE	t2	NULL	ref	t2_ac	t2_ac	5	sysbench.t1.a	1	33.33	Using index condition; Using where
1	SIMPLE	t3	NULL	ref	t3_bc	t3_bc	5	sysbench.t2.b	1	33.33	Using index condition

另外，**select *** 如果改成具体字段的话考虑覆盖索引 可能需要建立不同的索引。

2019-02-01

| 作者回复

验证的结果最有说服力

2019-02-01

36 | 为什么临时表可以重名？

2019-02-04 林晓斌



今天是大年三十，在开始我们今天的学习之前，我要先和你道一声春节快乐！

在上一篇文章中，我们在优化**join**查询的时候使用到了临时表。当时，我们是这么用的：

```
create temporary table temp_t like t1;
alter table temp_t add index(b);
insert into temp_t select * from t2 where b>=1 and b<=2000;
select * from t1 join temp_t on (t1.b=temp_t.b);
```

你可能会有疑问，为什么要用临时表呢？直接用普通表是不是也可以呢？

今天我们就从这个问题说起：临时表有哪些特征，为什么它适合这个场景？

这里，我需要先帮你厘清一个容易误解的问题：有的人可能会认为，临时表就是内存表。但是，这两个概念可是完全不同的。

- 内存表，指的是使用**Memory**引擎的表，建表语法是**create table ...engine=memory**。这种表的数据都保存在内存里，系统重启的时候会被清空，但是表结构还在。除了这两个特性看上去比较“奇怪”外，从其他的特征上看，它就是一个正常的表。
- 而临时表，可以使用各种引擎类型。如果是使用**InnoDB**引擎或者**MyISAM**引擎的临时表，写

数据的时候是写到磁盘上的。当然，临时表也可以使用**Memory**引擎。

弄清楚了内存表和临时表的区别以后，我们再来看看临时表有哪些特征。

临时表的特性

为了便于理解，我们来看下下面这个操作序列：

session A	session B
create temporary table t(c int)engine=myisam;	
	show create table t; (Table 't' doesn't exist)
create table t(id int primary key)engine=innodb; show create table t; //create temporary table t(c int)engine=myisam; show tables; //只显示普通表t	
	insert into t values(1); select * from t; //返回 1
select * from t; //Empty set	

图1 临时表特性示例

可以看到，临时表在使用上有以下几个特点：

1. 建表语法是**create temporary table ...**
2. 一个临时表只能被创建它的**session**访问，对其他线程不可见。所以，图中**session A**创建的临时表**t**，对于**session B**就是不可见的。
3. 临时表可以与普通表同名。
4. **session A**内有同名的临时表和普通表的时候，**show create**语句，以及增删改查语句访问的是临时表。
5. **show tables**命令不显示临时表。

由于临时表只能被创建它的**session**访问，所以在**session**结束的时候，会自动删除临时表。

也正是由于这个特性，临时表就特别适合我们文章开头的**join**优化这种场景。为什么呢？

原因主要包括以下两个方面：

1. 不同**session**的临时表是可以重名的，如果有多个**session**同时执行**join**优化，不需要担心表名重复导致建表失败的问题。
2. 不需要担心数据删除问题。如果使用普通表，在流程执行过程中客户端发生了异常断开，或者数据库发生异常重启，还需要专门来清理中间过程中生成的数据表。而临时表由于会自动回收，所以不需要这个额外的操作。

临时表的应用

由于不用担心线程之间的重名冲突，临时表经常会被用在复杂查询的优化过程中。其中，分库分表系统的跨库查询就是一个典型的使用场景。

一般分库分表的场景，就是要把一个逻辑上的大表分散到不同的数据库实例上。比如。将一个大表**ht**，按照字段**f**，拆分成**1024**个分表，然后分布到**32**个数据库实例上。如下图所示：

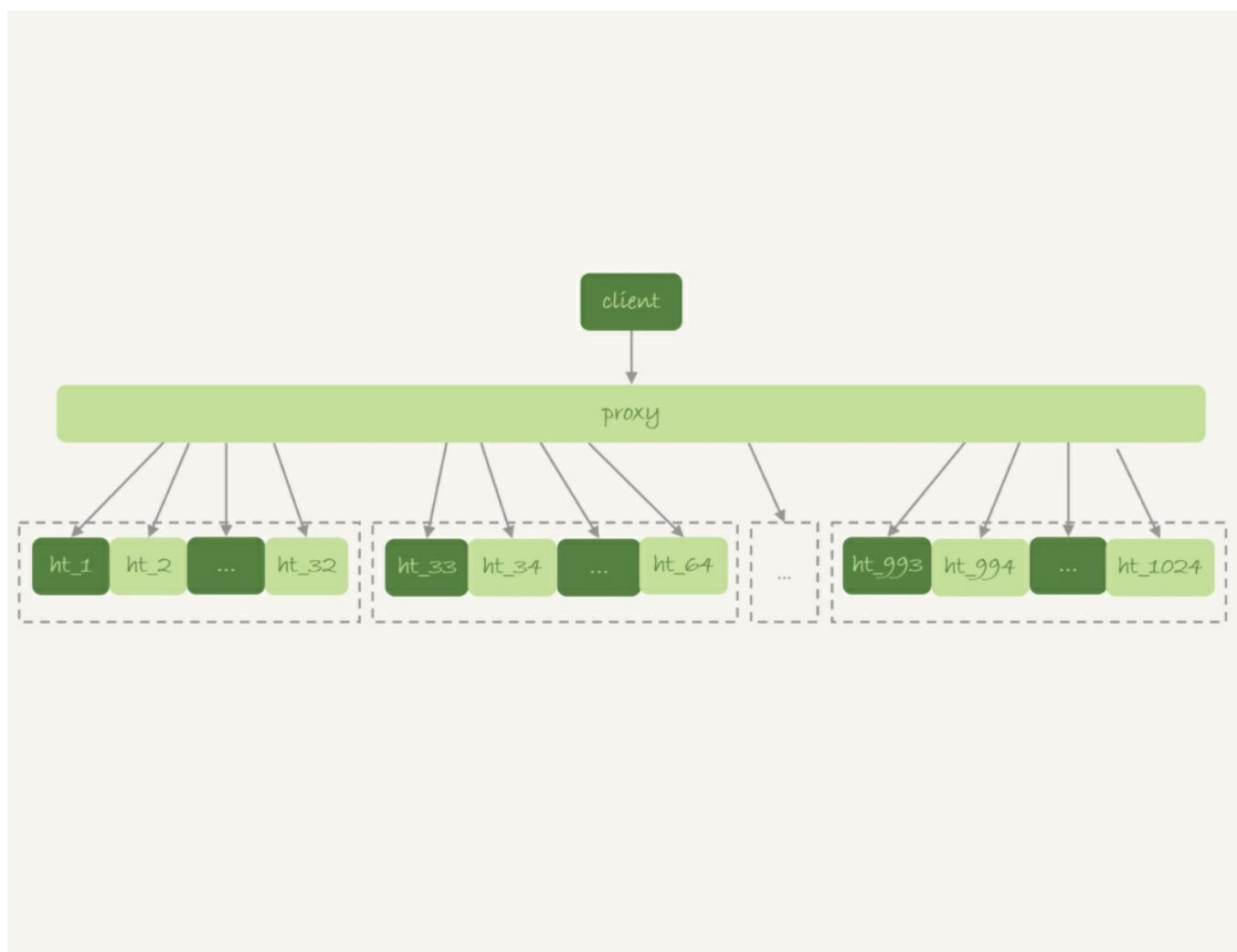


图2 分库分表简图

一般情况下，这种分库分表系统都有一个中间层**proxy**。不过，也有一些方案会让客户端直接连

接数据库，也就是没有proxy这一层。

在这个架构中，分区key的选择是以“减少跨库和跨表查询”为依据的。如果大部分的语句都会包含f的等值条件，那么就要用f做分区键。这样，在proxy这一层解析完SQL语句以后，就能确定将这条语句路由到哪个分表做查询。

比如下面这条语句：

```
select v from ht where f=N;
```

这时，我们就可以通过分表规则（比如， $N \% 1024$ ）来确认需要的数据被放在了哪个分表上。这种语句只需要访问一个分表，是分库分表方案最欢迎的语句形式了。

但是，如果这个表上还有另外一个索引k，并且查询语句是这样的：

```
select v from ht where k >= M order by t_modified desc limit 100;
```

这时候，由于查询条件里面没有用到分区字段f，只能到所有的分区中去查找满足条件的所有行，然后统一做order by的操作。这种情况下，有两种比较常用的思路。

第一种思路是，在proxy层的进程代码中实现排序。

这种方式的优势是处理速度快，拿到分库的数据以后，直接在内存中参与计算。不过，这个方案的缺点也比较明显：

1. 需要的开发工作量比较大。我们举例的这条语句还算是比较简单的，如果涉及到复杂的操作，比如group by，甚至join这样的操作，对中间层的开发能力要求比较高；
2. 对proxy端的压力比较大，尤其是很容易出现内存不够用和CPU瓶颈的问题。

另一种思路就是，把各个分库拿到的数据，汇总到一个MySQL实例的一个表中，然后在这个汇总实例上做逻辑操作。

比如上面这条语句，执行流程可以类似这样：

- 在汇总库上创建一个临时表temp_ht，表里包含三个字段v、k、t_modified；
- 在各个分库上执行

```
select v,k,t_modified from ht_x where k >= M order by t_modified desc limit 100;
```

- 把分库执行的结果插入到temp_ht表中；
- 执行

```
select v from temp_ht order by t_modified desc limit 100;
```

得到结果。

这个过程对应的流程图如下所示：

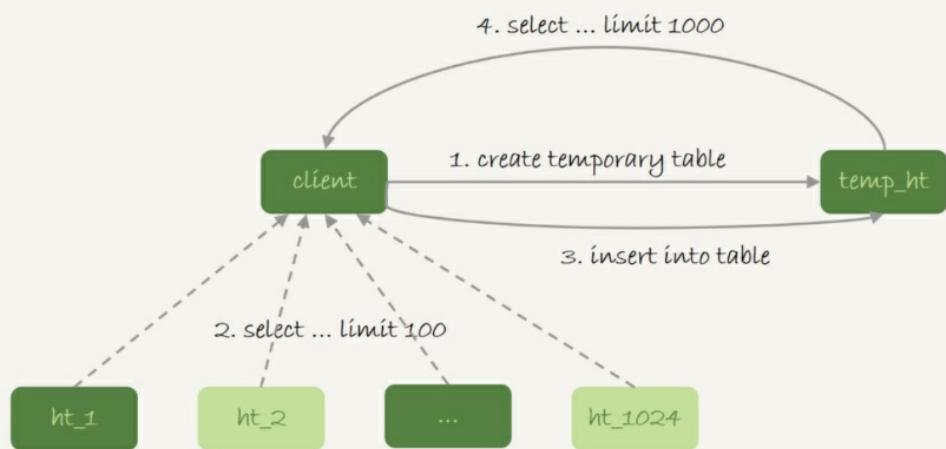


图3 跨库查询流程示意图

在实践中，我们往往你会发现每个分库的计算量都不饱和，所以会直接把临时表temp_ht放到32个分库中的某一个上。这时的查询逻辑与图3类似，你可以自己再思考一下具体的流程。

为什么临时表可以重名？

你可能会问，不同线程可以创建同名的临时表，这是怎么做到的呢？

接下来，我们就看一下这个问题。

我们在执行

```
create temporary table temp_t(id int primary key)engine=innodb;
```

这个语句的时候，MySQL要给这个InnoDB表创建一个frm文件保存表结构定义，还要有地方保存表数据。

这个frm文件放在临时文件目录下，文件名的后缀是.frm，前缀是“#sql{进程id}_{线程id}_序列号”。你可以使用select @@tmpdir命令，来显示实例的临时文件目录。

而关于表中数据的存放方式，在不同的MySQL版本中有着不同的处理方式：

- 在5.6以及之前的版本里，MySQL会在临时文件目录下创建一个相同前缀、以.ibd为后缀的文件，用来存放数据文件；
- 而从5.7版本开始，MySQL引入了一个临时文件表空间，专门用来存放临时文件的数据。因此，我们就不需要再创建.ibd文件了。

从文件名的前缀规则，我们可以看到，其实创建一个叫作t1的InnoDB临时表，MySQL在存储上认为我们创建的表名跟普通表t1是不同的，因此同一个库下面已经有普通表t1的情况下，还是可以再创建一个临时表t1的。

为了便于后面讨论，我先来举一个例子。

session A	session B
create temporary table t1 ... // #sql4d2_4_0.frm create temporary table t2 ... // #sql4d2_4_1.frm	
	create temporary table t1 ... // #sql4d2_5_0.frm

图4 临时表的表名

这个进程的进程号是1234，session A的线程id是4，session B的线程id是5。所以你看到了，session A和session B创建的临时表，在磁盘上的文件不会重名。

MySQL维护数据表，除了物理上要有文件外，内存里面也有一套机制区别不同的表，每个表都对应一个table_def_key。

- 一个普通表的table_def_key的值是由“库名+表名”得到的，所以如果你要在同一个库下创建两个同名的普通表，创建第二个表的过程中就会发现table_def_key已经存在了。
- 而对于临时表，table_def_key在“库名+表名”基础上，又加入了“server_id+thread_id”。

也就是说，**session A**和**sessionB**创建的两个临时表t1，它们的**table_def_key**不同，磁盘文件名也不同，因此可以并存。

在实现上，每个线程都维护了自己的临时表链表。这样每次**session**内操作表的时候，先遍历链表，检查是否有这个名字的临时表，如果有就优先操作临时表，如果没有再操作普通表；在**session**结束的时候，对链表里的每个临时表，执行“**DROP TEMPORARY TABLE +表名**”操作。

这时候你会发现，**binlog**中也记录了**DROP TEMPORARY TABLE**这条命令。你一定会觉得奇怪，临时表只在线程内自己可以访问，为什么需要写到**binlog**里面？

这，就需要说到主备复制了。

临时表和主备复制

既然写**binlog**，就意味着备库需要。

你可以设想一下，在主库上执行下面这个语句序列：

```
create table t_normal(id int primary key, c int)engine=innodb; /*Q1*/  
create temporary table temp_t like t_normal; /*Q2*/  
insert into temp_t values(1,1); /*Q3*/  
insert into t_normal select * from temp_t; /*Q4*/
```

如果关于临时表的操作都不记录，那么在备库就只有**create table t_normal**表和**insert into t_normal select * from temp_t**这两个语句的**binlog**日志，备库在执行到**insert into t_normal**的时候，就会报错“表**temp_t**不存在”。

你可能会说，如果把**binlog**设置为**row**格式就好了吧？因为**binlog**是**row**格式时，在记录**insert into t_normal**的**binlog**时，记录的是这个操作的数据，即：**write_row event**里面记录的逻辑是“插入一行数据（1,1）”。

确实是这样。如果当前的**binlog_format=row**，那么跟临时表有关的语句，就不会记录到**binlog**里。也就是说，只在**binlog_format=statement/mixed**的时候，**binlog**中才会记录临时表的操作。

这种情况下，创建临时表的语句会传到备库执行，因此备库的同步线程就会创建这个临时表。主库在线程退出的时候，会自动删除临时表，但是备库同步线程是持续在运行的。所以，这时候我们就需要在主库上再写一个**DROP TEMPORARY TABLE**传给备库执行。

之前有人问过我一个有趣的问题：MySQL在记录**binlog**的时候，不论是**create table**还是**alter table**语句，都是原样记录，甚至于连空格都不变。但是如果执行**drop table t_normal**，系统记录**binlog**就会写成：

```
DROP TABLE `t_normal` /* generated by server */
```

也就是改成了标准的格式。为什么要这么做呢？

现在你知道原因了，那就是：**drop table**命令是可以一次删除多个表的。比如，在上面的例子中，设置**binlog_format=row**，如果主库上执行 "drop table t_normal, temp_t"这个命令，那么**binlog**中就只能记录：

```
DROP TABLE `t_normal` /* generated by server */
```

因为备库上并没有表**temp_t**，将这个命令重写后再传到备库执行，才不会导致备库同步线程停止。

所以，**drop table**命令记录**binlog**的时候，就必须对语句做改写。“/* generated by server */”说明了这是一个被服务端改写过的命令。

说到主备复制，还有另外一个问题需要解决：主库上不同的线程创建同名的临时表是没关系的，但是传到备库执行是怎么处理的呢？

现在，我给你举个例子，下面的序列中实例**S**是**M**的备库。

	M上session A	M上session B	S上的应用日志线程
T1	create temporary table t1 ... ;		
T2			create temporary table t1 ... ;
T3		create temporary table t1...;	
T4			create temporary table t1...;

图5 主备关系中的临时表操作

主库**M**上的两个session创建了同名的临时表**t1**，这两个**create temporary table t1**语句都会被传到备库**S**上。

但是，备库的应用日志线程是共用的，也就是说要在应用线程里面先后执行这个**create**语句两次。（即使开了多线程复制，也可能被分配到从库的同一个**worker**中执行）。那么，这会不会导致同步线程报错？

显然是不会的，否则临时表就是一个**bug**了。也就是说，备库线程在执行的时候，要把这两个**t1**

表当做两个不同的临时表来处理。这，又是怎么实现的呢？

MySQL在记录binlog的时候，会把主库执行这个语句的线程id写到binlog中。这样，在备库的应用线程就能够知道执行每个语句的主库线程id，并利用这个线程id来构造临时表的table_def_key：

1. session A的临时表t1，在备库的table_def_key就是：库名+t1+“M的serverid”+“session A的thread_id”；
2. session B的临时表t1，在备库的table_def_key就是：库名+t1+“M的serverid”+“session B的thread_id”。

由于table_def_key不同，所以这两个表在备库的应用线程里面是不会冲突的。

小结

今天这篇文章，我和你介绍了临时表的用法和特性。

在实际应用中，临时表一般用于处理比较复杂的计算逻辑。由于临时表是每个线程自己可见的，所以不需要考虑多个线程执行同一个处理逻辑时，临时表的重名问题。在线程退出的时候，临时表也能自动删除，省去了收尾和异常处理的工作。

在binlog_format='row'的时候，临时表的操作不记录到binlog中，也省去了不少麻烦，这也成为你选择binlog_format时的一个考虑因素。

需要注意的是，我们上面说到的这种临时表，是用户自己创建的，也可以称为用户临时表。与它相对应的，就是内部临时表，在[第17篇文章](#)中我已经和你介绍过。

最后，我给你留下一个思考题吧。

下面的语句序列是创建一个临时表，并将其改名：

```
mysql> create temporary table temp_t(id int primary key)engine=innodb;
Query OK, 0 rows affected (0.01 sec)

mysql> alter table temp_t rename to temp_t2;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> rename table temp_t2 to temp_t3;
ERROR 1017 (HY000): Can't find file: './test/temp_t2.frm' (errno: 2 - No such file or directory)
```

图6 关于临时表改名的思考题

可以看到，我们可以使用alter table语法修改临时表的表名，而不能使用rename语法。你知道这是什么原因吗？

你可以把你的分析写在留言区，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也

欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，对于下面这个三个表的join语句，

```
select * from t1 join t2 on(t1.a=t2.a) join t3 on (t2.b=t3.b) where t1.c>=X and t2.c>=Y and t3.c>=Z;
```

如果改写成straight_join，要怎么指定连接顺序，以及怎么给三个表创建索引。

第一原则是要尽量使用BKA算法。需要注意的是，使用BKA算法的时候，并不是“先计算两个表join的结果，再跟第三个表join”，而是直接嵌套查询的。

具体实现是：在t1.c>=X、t2.c>=Y、t3.c>=Z这三个条件里，选择一个经过过滤以后，数据最少的那个表，作为第一个驱动表。此时，可能会出现如下两种情况。

第一种情况，如果选出来的是表t1或者t3，那剩下的部分就固定了。

1. 如果驱动表是t1，则连接顺序是t1->t2->t3，要在被驱动表字段创建上索引，也就是t2.a 和 t3.b上创建索引；
2. 如果驱动表是t3，则连接顺序是t3->t2->t1，需要在t2.b 和 t1.a上创建索引。

同时，我们还需要在第一个驱动表的字段c上创建索引。

第二种情况是，如果选出来的第一个驱动表是表t2的话，则需要评估另外两个条件的过滤效果。

总之，整体的思路就是，尽量让每一次参与join的驱动表的数据集，越小越好，因为这样我们的驱动表就会越小。

评论区留言点赞板：

@库淘淘 做了实验验证；

@poppy同学做了很不错的分析；

@dzkk 同学在评论中介绍了MariaDB支持的hash join，大家可以了解一下；

@老杨同志提了一个好问题，如果语句使用了索引a，结果还要对a排序，就不用MRR优化了，否则回表完还要增加额外的排序过程，得不偿失。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「」请朋友读，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



尘封

新年快乐

2019-02-04

1

| 作者回复

新年快乐

2019-02-04



亮

老师过年好呀，祝您猪年大吉，财源广进；老师咱们这个课结束后，再开一期好不好啊，没学够啊，这是我的新年愿望哦

2019-02-04

1

| 作者回复

新年快乐，共同进步

2019-02-04



辣椒

老师，不同线程可以使用同名的临时表，这个没有问题。但是如果在程序中，用的是连接池中的连接来操作的，而这些连接不会释放，和数据库保持长连接。这样使用临时表会有问题吗？

2019-02-07

0

| 作者回复

会，“临时表会自动回收”这个功能，主要用于“应用程序异常断开、MySQL异常重启”后，不需

要主动去删除表。

而平时正常使用的时候，用完删除，还是应该有的好习惯。¶

好问题，新年快乐~

2019-02-07



老杨同志

0

新年快乐，老师好勤奋！

有个问题，`insert into select`语句好像会给`select`的表加锁，如果没有索引，就锁全表，是不是这样？什么时候可以大胆的用这类语句？

2019-02-04

| 作者回复

新年好！

“`insert into select`语句好像会给`select`的表加锁，如果没有索引，就锁全表”，是的。

这类最好不要很大胆¶，如果不是业务急需的，从源表导出来再写到目标表也是好的。

后面第40篇会说到哈。

2019-02-05



慕塔

0

打卡 新年快乐¶¶

2019-02-04

| 作者回复

新年快乐、共同进步¶

好勤奋呀¶

2019-02-05



cheriston

0

老师辛苦了，大年三十还给我们分享技术，老师新年好¶.

2019-02-04

| 作者回复

同祝新年好，共同进步¶

2019-02-05



长杰

0

老师，新年快乐，万事如意！

2019-02-04

| 作者回复

新春快乐~

2019-02-05



杰

丁大大新春快乐

2019-02-04

0

| 作者回复

新年快乐 工作顺利~

2019-02-04



某、人

老师，新年快乐。由于自身原因，错过几期精彩的内容，年后上班以后在好好补补。

2019-02-04

0

| 作者回复

春节快乐 新年身体健康哈

2019-02-04



poppy

0

老师，新年快乐。

关于思考题，`alter table temp_t rename to temp_t2`,我的理解是mysql直接修改的是`table_def_key`，而对于`rename table temp_t2 to temp_t3`,mysql直接去mysql的data目录下该数据库的目录(例如老师实验用的应该是test数据库，所以对应的是test目录)下寻找名为`temp_t2.frm`的文件去修改名称，所以就出现了"Can't find file './test/temp_t2.frm'(errno: 2 - No such file or directory)

2019-02-04

| 作者回复

春节快乐

】

2019-02-04



亮

0

老师您好，在25课里面的置顶留言“6.表上无主键的情况(主库利用索引更改数据,备库回放只能用全表扫描,这种情况可以调整`slave_rows_search_algorithms`参数适当优化下)”

为啥会存在无主键的表呢，就算dba没创建主键，Innodb可以用rowid给自动建一个虚拟主键呀，这样不就是所有的表都有主键了吗？

2019-02-04

| 作者回复

用户没有显示指定主键的话，InnoDB引擎会自己创建一个隐藏的主键，但是这个主键对Server层是透明的，优化器用不上。

新年快乐~

2019-02-04

37 | 什么时候会使用内部临时表？

2019-02-06 林晓斌



今天是大年初二，在开始我们今天的学习之前，我要先和你道一声春节快乐！

在[第16](#)和[第34](#)篇文章中，我分别和你介绍了**sort buffer**、内存临时表和**join buffer**。这三个数据结构都是用来存放语句执行过程中的中间数据，以辅助**SQL**语句的执行的。其中，我们在排序的时候用到了**sort buffer**，在使用**join**语句的时候用到了**join buffer**。

然后，你可能会有这样的疑问，**MySQL**什么时候会使用内部临时表呢？

今天这篇文章，我就先给你举两个需要用到内部临时表的例子，来看看内部临时表是怎么工作的。然后，我们再来分析，什么情况下会使用内部临时表。

union 执行流程

为了便于量化分析，我用下面的表**t1**来举例。

```

create table t1(id int primary key, a int, b int, index(a));
delimiter ;;
create procedure idata()
begin
declare i int;

set i=1;
while(i<=1000)do
    insert into t1 values(i, i, i);
    set i=i+1;
end while;
end;;
delimiter ;
call idata();

```

然后，我们执行下面这条语句：

```
(select 1000 as f) union (select id from t1 order by id desc limit 2);
```

这条语句用到了**union**，它的语义是，取这两个子查询结果的并集。并集的意思就是这两个集合加起来，重复的行只保留一行。

下图是这个语句的**explain**结果。

```

mysql> explain (select 1000 as f) union (select id from t1 order by id desc limit 2);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | NULL | No tables used | | | | | | | | |
| 2 | UNION | t1 | NULL | index | NULL | PRIMARY | 4 | NULL | 2 | 100.00 | Using index |
| NULL | UNION RESULT | <union1,2> | NULL | ALL | NULL | NULL | NULL | NULL | NULL | NULL | Using temporary |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

图1 union语句explain 结果

可以看到：

- 第二行的**key=PRIMARY**，说明第二个子句用到了索引**id**。
- 第三行的**Extra**字段，表示在对子查询的结果集做**union**的时候，使用了临时表(**Using temporary**)。

这个语句的执行流程是这样的：

1. 创建一个内存临时表，这个临时表只有一个整型字段**f**，并且**f**是主键字段。

2. 执行第一个子查询，得到1000这个值，并存入临时表中。

3. 执行第二个子查询：

- 拿到第一行 $\text{id}=1000$ ，试图插入临时表中。但由于1000这个值已经存在于临时表了，违反了唯一性约束，所以插入失败，然后继续执行；
- 取到第二行 $\text{id}=999$ ，插入临时表成功。

4. 从临时表中按行取出数据，返回结果，并删除临时表，结果中包含两行数据分别是1000和999。

这个过程的流程图如下所示：

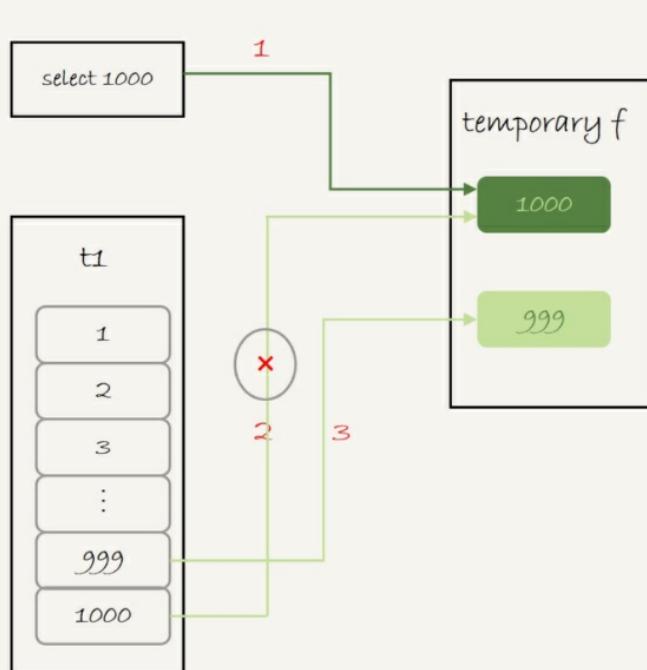


图 2 union 执行流程

可以看到，这里的内存临时表起到了暂存数据的作用，而且计算过程还用上了临时表主键 id 的唯一性约束，实现了union的语义。

顺便提一下，如果把上面这个语句中的union改成union all的话，就没有了“去重”的语义。这样执行的时候，就依次执行子查询，得到的结果直接作为结果集的一部分，发给客户端。因此也就不需要临时表了。

mysql> explain (select 1000 as f) union all (select id from t1 order by id desc limit 2);													
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra		
1	PRIMARY		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	No tables used	
2	UNION	t1	NULL	index	NULL	PRIMARY	4	NULL	2	100.00	Using index		

图3 union all的explain结果

可以看到，第二行的**Extra**字段显示的是**Using index**，表示只使用了覆盖索引，没有用临时表了。

group by 执行流程

另外一个常见的使用临时表的例子是**group by**，我们来看一下这个语句：

```
select id%10 as m, count(*) as c from t1 group by m;
```

这个语句的逻辑是把表t1里的数据，按照 **id%10** 进行分组统计，并按照m的结果排序后输出。它的**explain**结果如下：

mysql> explain select id%10 as m, count(*) as c from t1 group by m;													
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra		
1	SIMPLE	t1	NULL	index	PRIMARY,a	a	5	NULL	1000	100.00	Using index; Using temporary; Using filesort		

图4 group by 的explain结果

在**Extra**字段里面，我们可以看到三个信息：

- **Using index**, 表示这个语句使用了覆盖索引，选择了索引a，不需要回表；
- **Using temporary**, 表示使用了临时表；
- **Using filesort**, 表示需要排序。

这个语句的执行流程是这样的：

1. 创建内存临时表，表里有两个字段**m**和**c**，主键是**m**；
2. 扫描表t1的索引a，依次取出叶子节点上的**id**值，计算**id%10**的结果，记为**x**；
 - 如果临时表中没有主键为**x**的行，就插入一个记录(**x,1**)；
 - 如果表中有主键为**x**的行，就将**x**这一行的**c**值加1；
3. 遍历完成后，再根据字段**m**做排序，得到结果集返回给客户端。

这个流程的执行图如下：

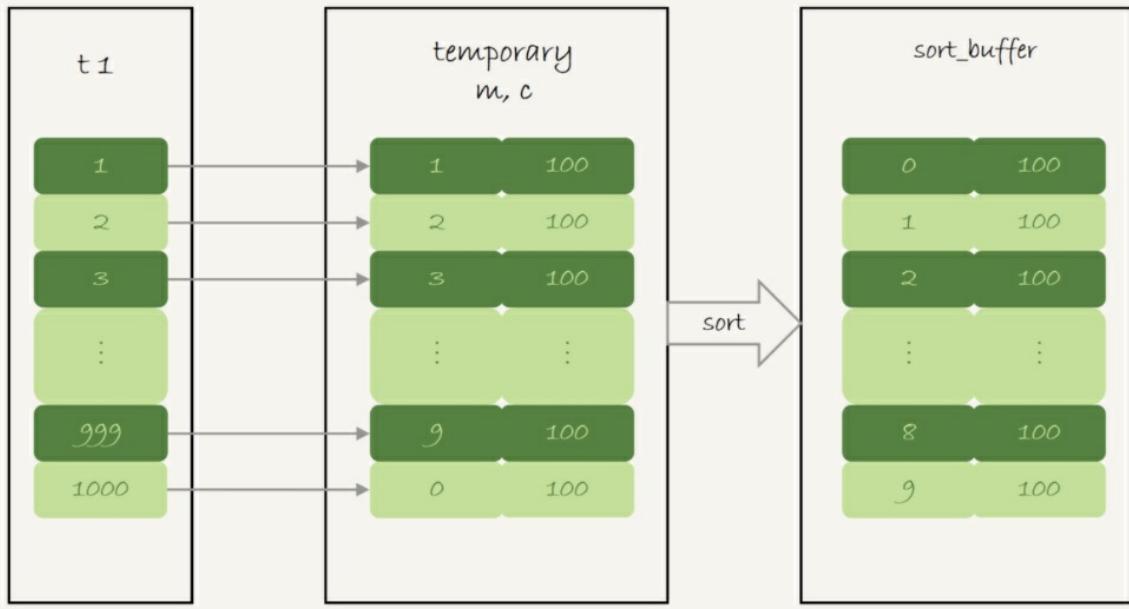


图5 group by执行流程

图中最后一步，对内存临时表的排序，在[第17篇文章](#)中已经有过介绍，我把图贴过来，方便你回顾。

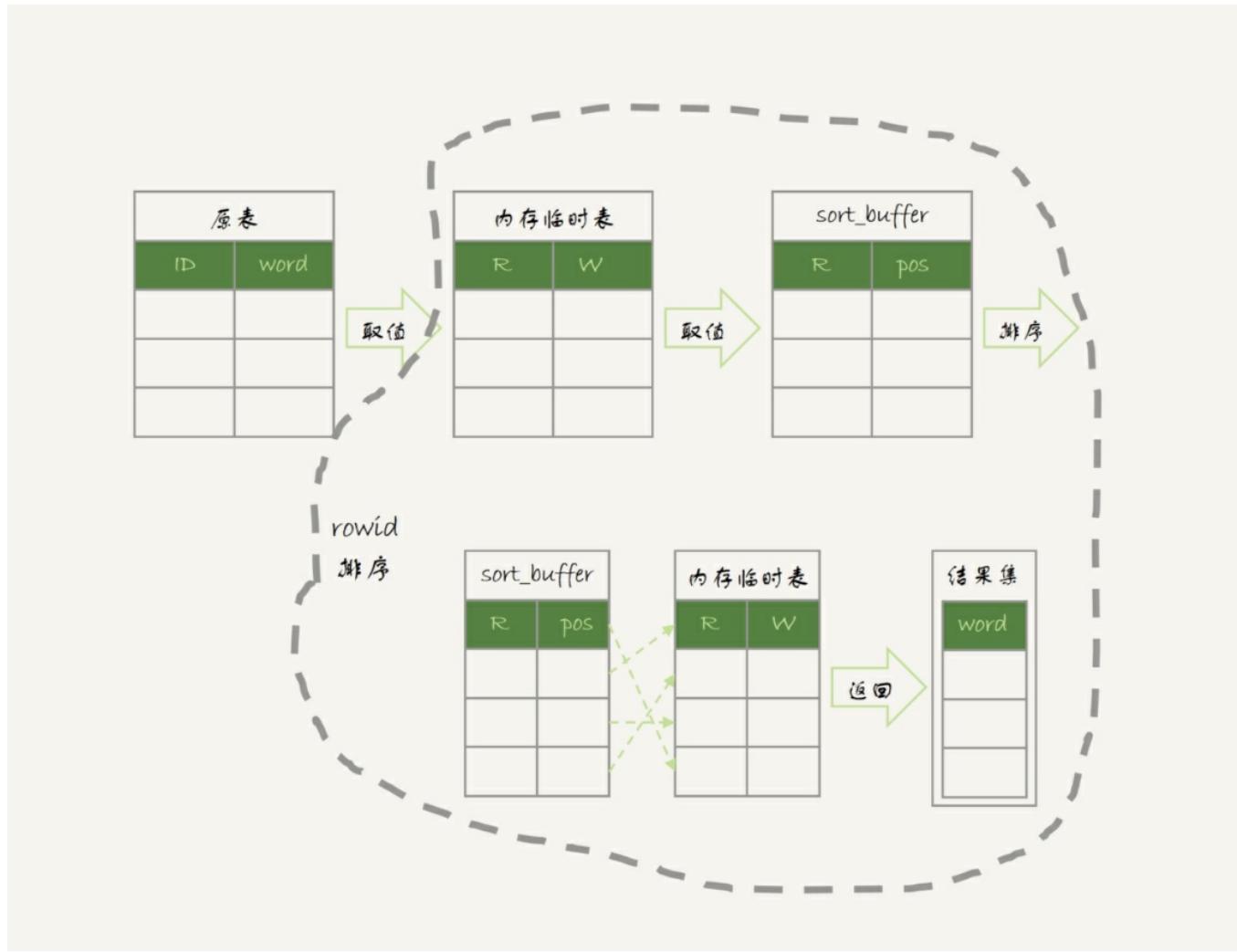


图6 内存临时表排序流程

其中，临时表的排序过程就是图6中虚线框内的过程。

接下来，我们再看一下这条语句的执行结果：

```
mysql> select id%10 as m, count(*) as c from t1 group by m;
+---+---+
| m | c |
+---+---+
| 0 | 100 |
| 1 | 100 |
| 2 | 100 |
| 3 | 100 |
| 4 | 100 |
| 5 | 100 |
| 6 | 100 |
| 7 | 100 |
| 8 | 100 |
| 9 | 100 |
+---+---+
```

图 7 group by 执行结果

如果你的需求并不需要对结果进行排序，那你在SQL语句末尾增加order by null，也就是改成：

```
select id%10 as m, count(*) as c from t1 group by m order by null;
```

这样就跳过了最后排序的阶段，直接从临时表中取数据返回。返回的结果如图8所示。

```
mysql> select id%10 as m, count(*) as c from t1 group by m order by null;
+---+---+
| m | c |
+---+---+
| 1 | 100|
| 2 | 100|
| 3 | 100|
| 4 | 100|
| 5 | 100|
| 6 | 100|
| 7 | 100|
| 8 | 100|
| 9 | 100|
| 0 | 100|
+---+---+
10 rows in set (0.00 sec)
```

图8 group + order by null 的结果（内存临时表）

由于表t1中的id值是从1开始的，因此返回的结果集中第一行是m=1；扫描到id=10的时候才插入m=0这一行，因此结果集里最后一行才是m=0。

这个例子里由于临时表只有10行，内存可以放得下，因此全程只使用了内存临时表。但是，内存临时表的大小是有限制的，参数tmp_table_size就是控制这个内存大小的，默认是16M。

如果我执行下面这个语句序列：

```
set tmp_table_size=1024;
select id%100 as m, count(*) as c from t1 group by m order by null limit 10;
```

把内存临时表的大小限制为最大1024字节，并把语句改成id % 100，这样返回结果里有100行数据。但是，这时的内存临时表大小不够存下这100行数据，也就是说，执行过程中会发现内存临时表大小到达了上限（1024字节）。

那么，这时候就会把内存临时表转成磁盘临时表，磁盘临时表默认使用的引擎是InnoDB。这

时，返回的结果如图9所示。

m	c
0	10
1	10
2	10
3	10
4	10
5	10
6	10
7	10
8	10
9	10

10 rows in set (0.01 sec)

图9 group + order by null 的结果（磁盘临时表）

如果这个表t1的数据量很大，很可能这个查询需要的磁盘临时表就会占用大量的磁盘空间。

group by 优化方法 --索引

可以看到，不论是使用内存临时表还是磁盘临时表，group by逻辑都需要构造一个带唯一索引的表，执行代价都是比较高的。如果表的数据量比较大，上面这个group by语句执行起来就会很慢，我们有什么优化的方法呢？

要解决group by语句的优化问题，你可以先想一下这个问题：执行group by语句为什么需要临时表？

group by的语义逻辑，是统计不同的值出现的个数。但是，由于每一行的id%100的结果是无序的，所以我们就需要有一个临时表，来记录并统计结果。

那么，如果扫描过程中可以保证出现的数据是有序的，是不是就简单了呢？

假设，现在有一个类似图10的这么一个数据结构，我们来看看group by可以怎么做。

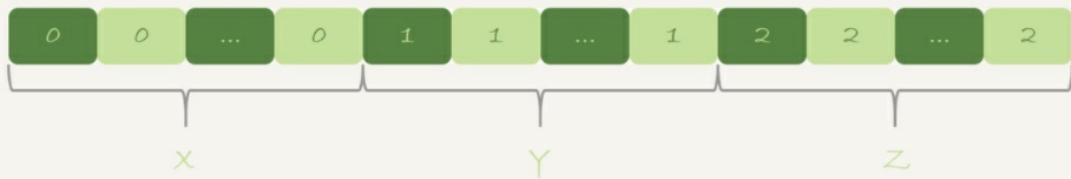


图10 group by算法优化-有序输入

可以看到，如果可以确保输入的数据是有序的，那么计算**group by**的时候，就只需要从左到右，顺序扫描，依次累加。也就是下面这个过程：

- 当碰到第一个1的时候，已经知道累积了X个0，结果集里的第一行就是(0,X)；
- 当碰到第一个2的时候，已经知道累积了Y个1，结果集里的第一行就是(1,Y)；

按照这个逻辑执行的话，扫描到整个输入的数据结束，就可以拿到**group by**的结果，不需要临时表，也不需要再额外排序。

你一定想到了，InnoDB的索引，就可以满足这个输入有序的条件。

在MySQL 5.7版本支持了**generated column**机制，用来实现列数据的关联更新。你可以用下面的方法创建一个列Z，然后在Z列上创建一个索引（如果是MySQL 5.6及之前的版本，你也可以创建普通列和索引，来解决这个问题）。

```
alter table t1 add column z int generated always as(id % 100), add index(z);
```

这样，索引Z上的数据就是类似图10这样有序的了。上面的**group by**语句就可以改成：

```
select z, count(*) as c from t1 group by z;
```

优化后的group by语句的explain结果，如下图所示：

```
mysql> explain select z , count(*) as c from t1 group by z;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | t1   | NULL       | index | z           | z    | 5        | NULL | 1000 | 100.00  | Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图11 group by 优化的explain结果

从Extra字段可以看到，这个语句的执行不再需要临时表，也不需要排序了。

group by优化方法 --直接排序

所以，如果可以通过加索引来完成group by逻辑就再好不过了。但是，如果碰上不适合创建索引的场景，我们还是要老老实实做排序的。那么，这时候的group by要怎么优化呢？

如果我们明明知道，一个group by语句中需要放到临时表上的数据量特别大，却还是要按照“先放到内存临时表，插入一部分数据后，发现内存临时表不够用了再转成磁盘临时表”，看上去就有点儿傻。

那么，我们就会想了，MySQL有没有让我们直接走磁盘临时表的方法呢？

答案是，有的。

在group by语句中加入SQL_BIG_RESULT这个提示（hint），就可以告诉优化器：这个语句涉及的数据量很大，请直接用磁盘临时表。

MySQL的优化器一看，磁盘临时表是B+树存储，存储效率不如数组来得高。所以，既然你告诉我数据量很大，那从磁盘空间考虑，还是直接用数组来存吧。

因此，下面这个语句

```
select SQL_BIG_RESULT id%100 as m, count(*) as c from t1 group by m;
```

的执行流程就是这样的：

1. 初始化sort_buffer，确定放入一个整型字段，记为m；
2. 扫描表t1的索引a，依次取出里面的id值，将 id%100的值存入sort_buffer中；
3. 扫描完成后，对sort_buffer的字段m做排序（如果sort_buffer内存不够用，就会利用磁盘临时文件辅助排序）；

4. 排序完成后，就得到了一个有序数组。

根据有序数组，得到数组里面的不同值，以及每个值的出现次数。这一步的逻辑，你已经从前面的图10中了解过了。

下面两张图分别是执行流程图和执行explain命令得到的结果。

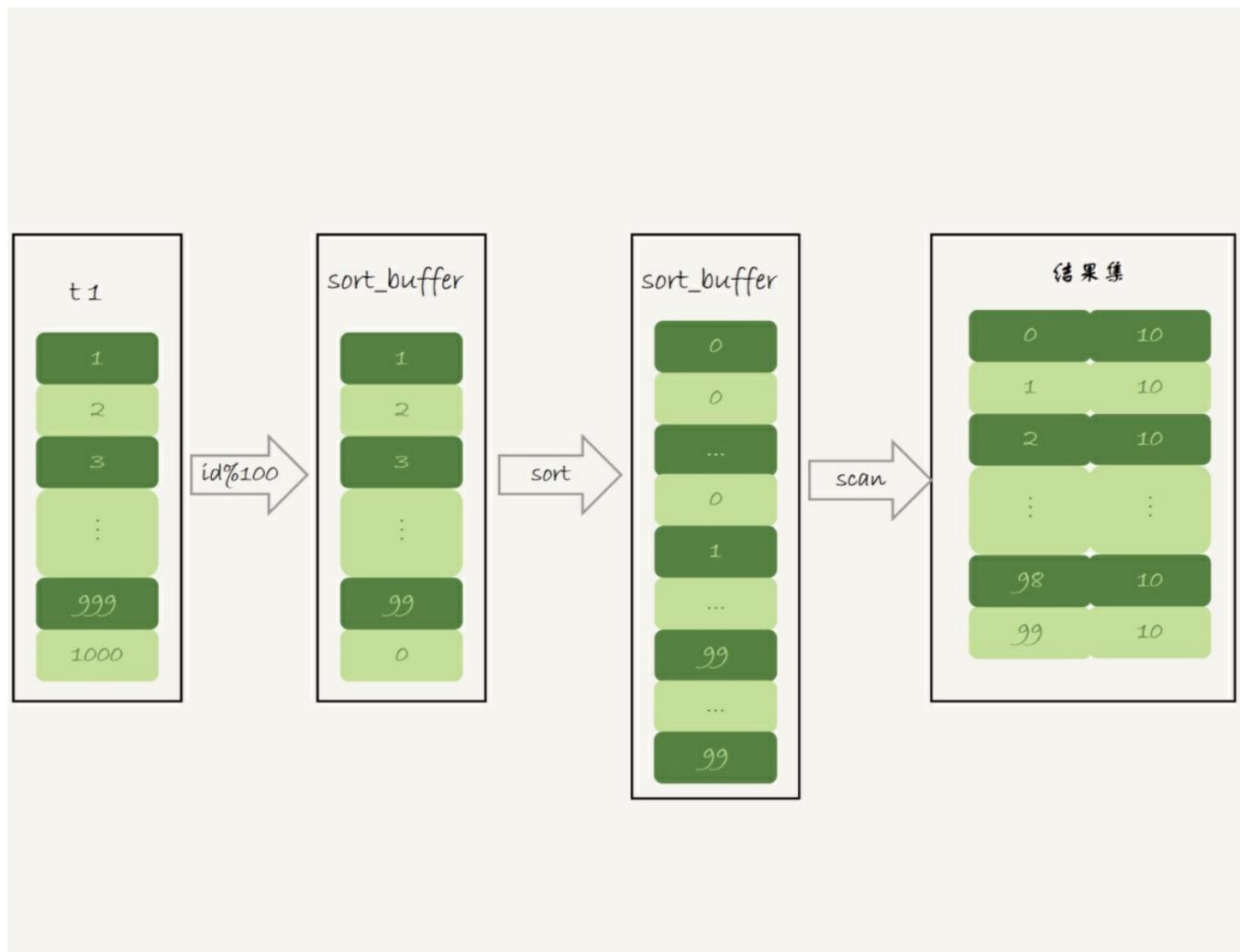


图12 使用 `SQL_BIG_RESULT`的执行流程图

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t1	NULL	index	PRIMARY,a	a	5	NULL	1000	100.00	Using index; Using filesort

图13 使用 `SQL_BIG_RESULT`的explain 结果

从`Extra`字段可以看到，这个语句的执行没有再使用临时表，而是直接用了排序算法。

基于上面的`union`、`union all`和`group by`语句的执行过程的分析，我们来回答文章开头的问题：

MySQL什么时候会使用内部临时表？

1. 如果语句执行过程可以一边读数据，一边直接得到结果，是不需要额外内存的，否则就需要

额外的内存，来保存中间结果；

2. `join_buffer`是无序数组，`sort_buffer`是有序数组，临时表是二维表结构；
3. 如果执行逻辑需要用到二维表特性，就会优先考虑使用临时表。比如我们的例子中，`union`需要用到唯一索引约束，`group by`还需要用到另外一个字段来存累积计数。

小结

通过今天这篇文章，我重点和你讲了`group by`的几种实现算法，从中可以总结一些使用的指导原则：

1. 如果对`group by`语句的结果没有排序要求，要在语句后面加`order by null`；
2. 尽量让`group by`过程用上表的索引，确认方法是`explain`结果里没有`Using temporary`和`Using filesort`；
3. 如果`group by`需要统计的数据量不大，尽量只使用内存临时表；也可以通过适当调大`tmp_table_size`参数，来避免用到磁盘临时表；
4. 如果数据量实在太大，使用`SQL_BIG_RESULT`这个提示，来告诉优化器直接使用排序算法得到`group by`的结果。

最后，我给你留下一个思考题吧。

文章中图8和图9都是`order by null`，为什么图8的返回结果里面，0是在结果集的最后一行，而图9的结果里面，0是在结果集的第一行？

你可以把你的分析写在留言区里，我会在下一篇文章和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是：为什么不能用`rename`修改临时表的改名。

在实现上，执行`rename table`语句的时候，要求按照“库名/表名.frm”的规则去磁盘找文件，但是临时表在磁盘上的frm文件是放在`tmpdir`目录下的，并且文件名的规则是“#sql{进程id}_{线程id}_序列号.frm”，因此会报“找不到文件名”的错误。

评论区留言点赞板：

| @poppy 同学，通过执行语句的报错现象推测了这个实现过程。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「」请朋友读，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



老杨同志

1

请教一个问题：如果只需要去重，不需要执行聚合函数，`distinct` 和 `group by` 那种效率高一些呢？

课后习题：

图8，把统计结果存内存临时表，不排序。`id`是从1到1000，模10的结果顺序就是1、2、3、4、5。。。

图9，老师把`tmp_table_size`改小了，内存临时表装不下，改用磁盘临时表。根据老师讲的流程，`id`取模的结果，排序后存入临时表，临时的数据应该是0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,2,

.....

从这个磁盘临时表读取数据汇总的结果的顺序就是0,1,2,3,4,5。。。

2019-02-06

| 作者回复

新年好

好问题，我加到后面文章中。

简单说下结论，只需要去重的话，如果没有`limit`，是一样的；
有`limit`的话，`distinct` 快些。

漂亮的回答

2019-02-07



Long

老师，新年好！:-)

0

有几个版本差异的问题：

(1) 图1中的执行计划应该是5.7版本以后的吧，貌似没找到说在哪个环境，我在5.6和5.7分别测试了，`id = 2`的那个`rows`，在5.6版本（5.6.26）是1000，在5.7版本是2行。应该是5.7做的优化吧？

(2) 图 9 `group + order by null` 的结果（此盘临时表），这里面mysql5.6里面执行的结果是（1， 10），（2， 10）...(10, 10)，执行计划都是只有一样，没找到差异。

跟踪下了下`optimizer trace`，发现问题应该是在临时表空间满的时候，mysql5.7用的是：`converting_tmp_table_to_ondisk "location": "disk (InnoDB)"`，而mysql 5.6用的是`converting_tmp_table_to_myisam "location": "disk (MyISAM)"`的原因导致的。

查了下参数：

`default_tmp_storage_engine`。（5.6, 5.7当前值都是`innodb`）

`internal_tmp_disk_storage_engine`（只有5.7有这个参数，当前值是`innodb`），5.6应该是默认磁盘临时表就是MyISAM引擎的了，由于本地测试环境那个临时表的目录下找不到临时文件，也没法继续分析了。。。

至于为什么MySQL 5.6中结果展示`m`字段不是0-9而是1-10，还得请老师帮忙解答下了。

还有几个小问题，为了方便解答，序号统一了：

(3) 在阅读mysql执行计划的时候，看了网上有很多说法，也参考了mysql官网对`id` (`select_id`) 的解释：

`id` (JSON name: `select_id`)

The SELECT identifier. This is the sequential number of the SELECT within the query. (感觉这个读起来也有点歧义，这个`sequential`字面解释感觉只有顺序的号码，并咩有说执行顺序)

比如图1，文中解释就是从ID小的往大的执行的，网上有很多其他说法，有的是说ID从大到小执行，遇到ID一样的，就从上往下执行。有的说是从小往大顺序执行。不知道老师是否可以官方讲解下。

(4) 我发现想搞懂一个原理，并且讲清楚让别人明白，真的是很有难度，非常感谢老师的分享。这次专栏结束，还会推出的新的专栏吗？非常期待。

2019-02-10



Laputa

老师好，文中说的不需要排序为什么不直接把`order by`去掉而是写`order by null`

0

2019-02-08

| 作者回复

MySQL 语义上这么定义的...

2019-02-08



HuaMax

0

课后题解答。图8是用内存临时表，文中已经提到，是按照表t1的索引a顺序取出数据，模10得0的id是最后一行；图9是用硬盘临时表，默认用innodb的索引，主键是id%10，因此存入硬盘后再按主键树顺序取出，0就排到第一行了。

2019-02-07



Li Shunduo

0

请问Group By部分的第一个语句 `explain select id%10 as m, count(*) as c from t1 group by m;` 为什么选择的是索引a，而不是primary key？如果字段a上有空值，使用索引a岂不是就不能取到所有的id值了？

2019-02-07

| 作者回复

因为索引c的信息也足够，而且比主键索引小，使用索引c更会好。

“如果字段a上有空值，使用索引a岂不是就不能取到所有的id值了？”，不会的

2019-02-07



牛牛

0

新年快乐~、感谢有您~^_^~

2019-02-06

| 作者回复

新年快乐~

2019-02-07



poppy

0

老师，春节快乐，过年还在更新，辛苦辛苦。

关于思考题，我的理解是图8中的查询是使用了内存临时表，存储的顺序就是id%10的值的插入顺序，而图9中的查询，由于内存临时表大小无法满足，所以使用了磁盘临时表，对于InnoDB来说，就是对应B+树这种数据结构，这里会按照id%100(即m)的大小顺序来存储的，所以返回的结果当然也是有序的

2019-02-06

| 作者回复

新年好~

0

2019-02-07



张八百

0

春节快乐，老师。谢谢你让我学到不少知识

2019-02-06

| 作者回复

新年快乐

2019-02-06



某、人

老师春节快乐，辛苦了

2019-02-06

0

| 作者回复

春节快乐，

2019-02-06



长杰

0

图九使用的是磁盘临时表，磁盘临时表使用的引擎是innodb，innodb是索引组织表，按主键顺序存储数据，所以是按照m字段有序的。

2019-02-06

| 作者回复

||

春节快乐

2019-02-06

38 | 都说InnoDB好，那还要不要使用Memory引擎？

2019-02-08 林晓斌



我在上一篇文章末尾留给你的问题是：两个`group by`语句都用了`order by null`，为什么使用内存临时表得到的语句结果里，0这个值在最后一行；而使用磁盘临时表得到的结果里，0这个值在第一行？

今天我们就来看看，出现这个问题的原因吧。

内存表的数据组织结构

为了便于分析，我来把这个问题简化一下，假设有以下的两张表t1和t2，其中表t1使用Memory引擎，表t2使用InnoDB引擎。

```
create table t1(id int primary key, c int) engine=Memory;
create table t2(id int primary key, c int) engine=innodb;
insert into t1 values(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(0,0);
insert into t2 values(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(0,0);
```

然后，我分别执行`select * from t1`和`select * from t2`。

mysql> select * from t1;		mysql> select * from t2;	
id	c	id	c
1	1	0	0
2	2	1	1
3	3	2	2
4	4	3	3
5	5	4	4
6	6	5	5
7	7	6	6
8	8	7	7
9	9	8	8
0	0	9	9

10 rows in set (0.00 sec) 10 rows in set (0.00 sec)

图1 两个查询结果-0的位置

可以看到，内存表t1的返回结果里面0在最后一行，而InnoDB表t2的返回结果里0在第一行。

出现这个区别的原因，要从这两个引擎的主键索引的组织方式说起。

表t2用的是InnoDB引擎，它的主键索引id的组织方式，你已经很熟悉了：InnoDB表的数据就放在主键索引树上，主键索引是B+树。所以表t2的数据组织方式如下图所示：

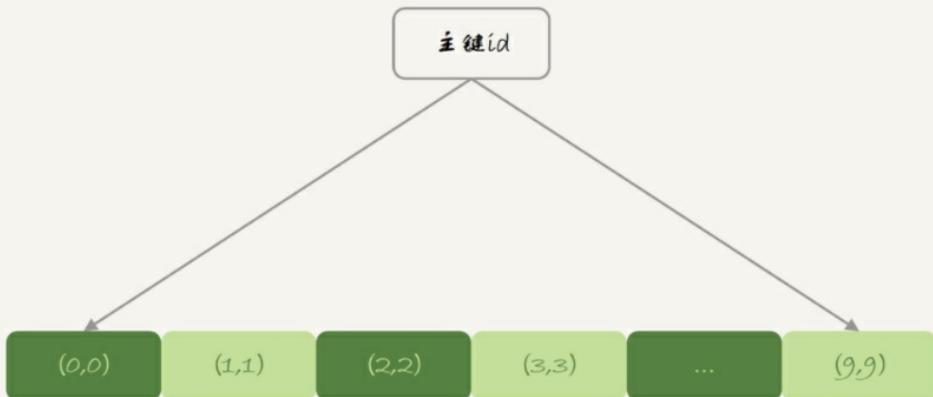


图2 表t2的数据组织

主键索引上的值是有序存储的。在执行**select ***的时候，就会按照叶子节点从左到右扫描，所以得到的结果里，0就出现在第一行。

与InnoDB引擎不同，Memory引擎的数据和索引是分开的。我们来看一下表t1中的数据内容。

主键id

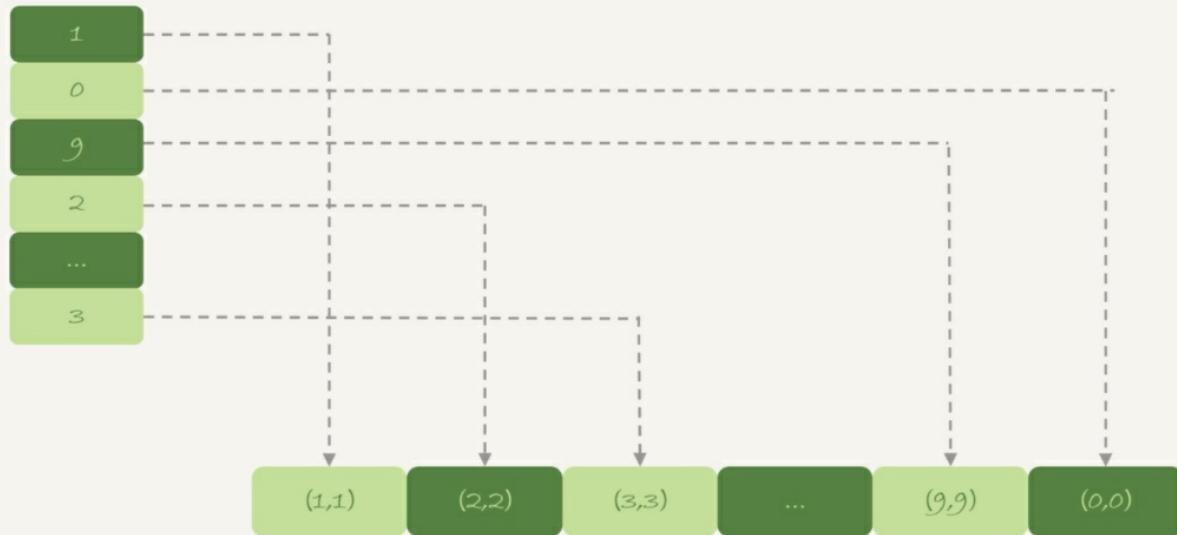


图3 表t1 的数据组织

可以看到，内存表的数据部分以数组的方式单独存放，而主键id索引里，存的是每个数据的位置。主键id是hash索引，可以看到索引上的key并不是有序的。

在内存表t1中，当我执行`select *`的时候，走的是全表扫描，也就是顺序扫描这个数组。因此，0就是最后一个被读到，并放入结果集的数据。

可见，InnoDB和Memory引擎的数据组织方式是不同的：

- InnoDB引擎把数据放在主键索引上，其他索引上保存的是主键id。这种方式，我们称之为索引组织表（Index Organized Table）。
- 而Memory引擎采用的是把数据单独存放，索引上保存数据位置的数据组织形式，我们称之为堆组织表（Heap Organized Table）。

从中我们可以看出，这两个引擎的一些典型不同：

1. InnoDB表的数据总是有序存放的，而内存表的数据就是按照写入顺序存放的；
2. 当数据文件有空洞的时候，InnoDB表在插入新数据的时候，为了保证数据有序性，只能在固定的位置写入新值，而内存表找到空位就可以插入新值；

3. 数据位置发生变化的时候，InnoDB表只需要修改主键索引，而内存表需要修改所有索引；
4. InnoDB表用主键索引查询时需要走一次索引查找，用普通索引查询的时候，需要走两次索引查找。而内存表没有这个区别，所有索引的“地位”都是相同的。
5. InnoDB支持变长数据类型，不同记录的长度可能不同；内存表不支持Blob和Text字段，并且即使定义了varchar(N)，实际也当作char(N)，也就是固定长度字符串来存储，因此内存表的每行数据长度相同。

由于内存表的这些特性，每个数据行被删除以后，空出的这个位置都可以被接下来要插入的数据复用。比如，如果要在表t1中执行：

```
delete from t1 where id=5;  
insert into t1 values(10,10);  
select * from t1;
```

就会看到返回结果里，id=10这一行出现在id=4之后，也就是原来id=5这行数据的位置。

需要指出的是，表t1的这个主键索引是哈希索引，因此如果执行范围查询，比如

```
select * from t1 where id<5;
```

是用不上主键索引的，需要走全表扫描。你可以借此再回顾下[第4篇文章](#)的内容。那如果要让内存表支持范围扫描，应该怎么办呢？

hash索引和B-Tree索引

实际上，内存表也是支B-Tree索引的。在id列上创建一个B-Tree索引，SQL语句可以这么写：

```
alter table t1 add index a_btreet_index using btree (id);
```

这时，表t1的数据组织形式就变成了这样：

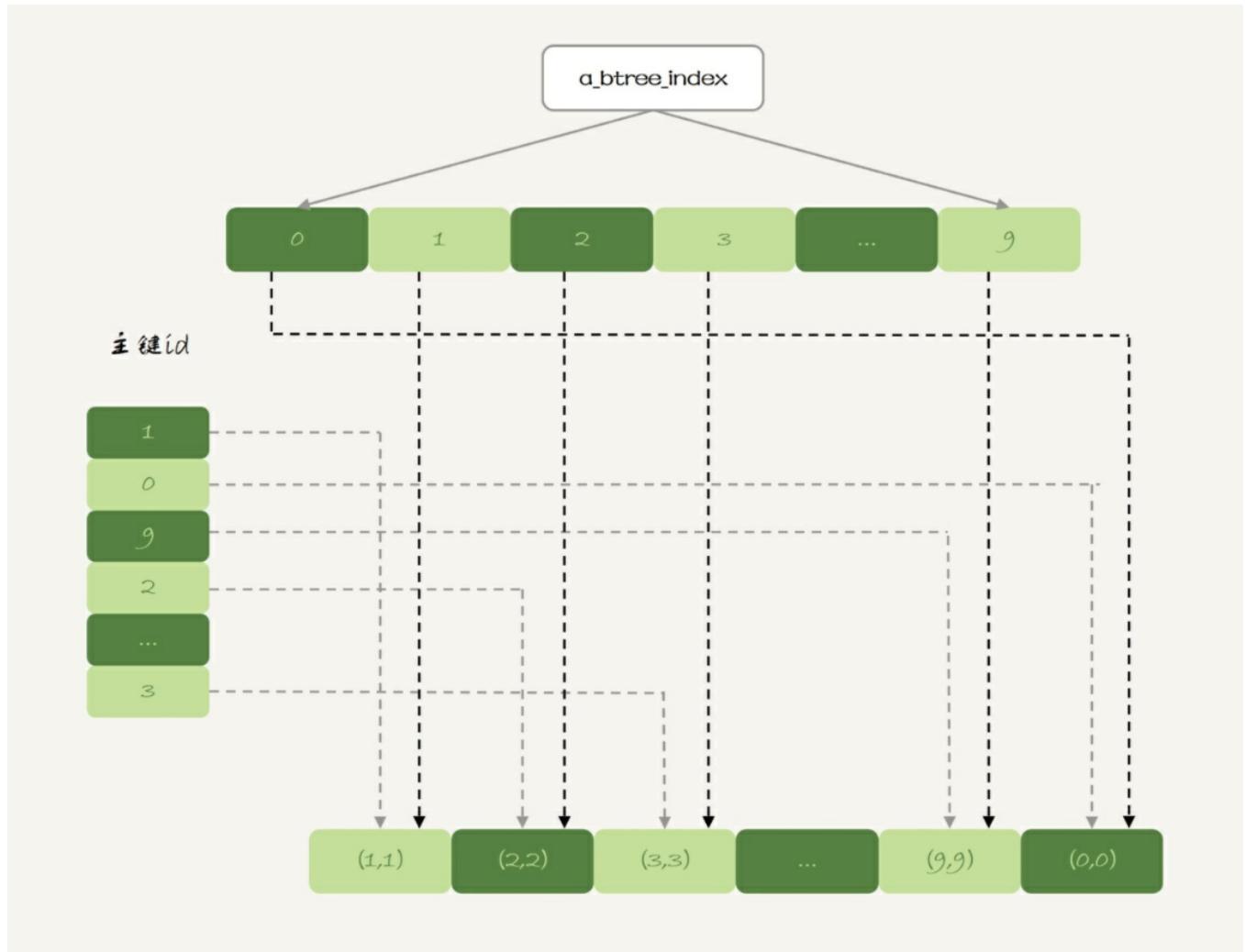


图4 表t1的数据组织—增加B-Tree索引

新增的这个**B-Tree**索引你看着就眼熟了，这跟InnoDB的**b+树**索引组织形式类似。

作为对比，你可以看一下这下面这两个语句的输出：

```

mysql> select * from t1 where id<5;
+---+---+
| id | c   |
+---+---+
| 0  | 0   |
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
| 4  | 4   |
+---+---+
5 rows in set (0.00 sec)

mysql> select * from t1 force index(primary) where id<5;
+---+---+
| id | c   |
+---+---+
| 1  | 1   |
| 2  | 2   |
| 3  | 3   |
| 4  | 4   |
| 0  | 0   |
+---+---+
5 rows in set (0.00 sec)

```

图5 使用B-Tree和hash索引查询返回结果对比

可以看到，执行`select * from t1 where id<5`的时候，优化器会选择B-Tree索引，所以返回结果是0到4。使用`force index`强行使用主键id这个索引，`id=0`这一行就在结果集的最末尾了。

其实，一般在我们的印象中，内存表的优势是速度快，其中的一个原因就是Memory引擎支持hash索引。当然，更重要的原因是，内存表的所有数据都保存在内存，而内存的读写速度总是比磁盘快。

但是，接下来我要跟你说明，为什么我不建议你在生产环境上使用内存表。这里的原因主要包括两个方面：

1. 锁粒度问题；
2. 数据持久化问题。

内存表的锁

我们先来说说内存表的锁粒度问题。

内存表不支持行锁，只支持表锁。因此，一张表只要有更新，就会堵住其他所有在这个表上的读

写操作。

需要注意的是，这里的表锁跟之前我们介绍过的MDL锁不同，但都是表级的锁。接下来，我通过下面这个场景，跟你模拟一下内存表的表级锁。

session A	session B	session C
update t1 set id=sleep(50) where id=1;		
	select * from t1 where id=2; (wait 50s)	
		show processlist;

图6 内存表的表锁--复现步骤

在这个执行序列里，**session A**的**update**语句要执行50秒，在这个语句执行期间**session B**的查询会进入锁等待状态。**session C**的**show processlist**结果输出如下：

mysql> show processlist;						
Id	User	Host	db	Command	Time	State
4	root	localhost:28350	test	Query	3	User sleep
5	root	localhost:28452	test	Query	1	Waiting for table level lock
6	root	localhost:28498	test	Query	0	starting

图7 内存表的表锁--结果

跟行锁比起来，表锁对并发访问的支持不够好。所以，内存表的锁粒度问题，决定了它在处理并发事务的时候，性能也不会太好。

数据持久性问题

接下来，我们再看看数据持久性的问题。

数据放在内存中，是内存表的优势，但也是一个劣势。因为，数据库重启的时候，所有的内存表都会被清空。

你可能会说，如果数据库异常重启，内存表被清空也就清空了，不会有什么问题啊。但是，在高可用架构下，内存表的这个特点简直可以当做bug来看待了。为什么这么说呢？

我们先看看**M-S**架构下，使用内存表存在的问题。

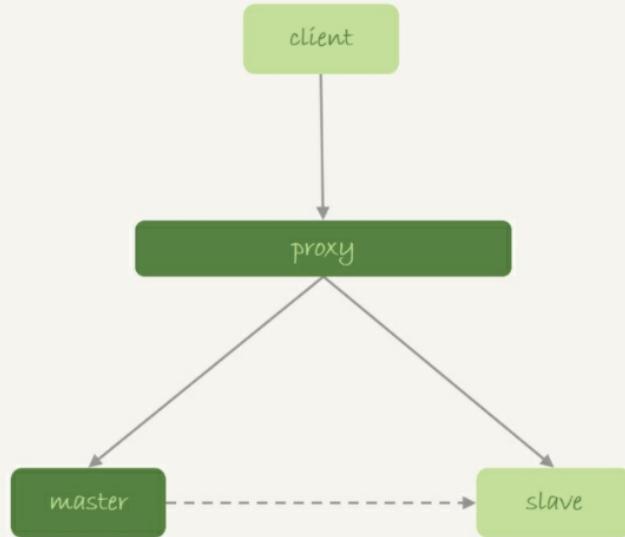


图8 M-S基本架构

我们来看一下下面这个时序：

1. 业务正常访问主库；
2. 备库硬件升级，备库重启，内存表t1内容被清空；
3. 备库重启后，客户端发送一条update语句，修改表t1的数据行，这时备库应用线程就会报错“找不到要更新的行”。

这样就会导致主备同步停止。当然，如果这时候发生主备切换的话，客户端会看到，表t1的数据“丢失”了。

在图8中这种有proxy的架构里，大家默认主备切换的逻辑是由数据库系统自己维护的。这样对客户端来说，就是“网络断开，重连之后，发现内存表数据丢失了”。

你可能说这还好啊，毕竟主备发生切换，连接会断开，业务端能够感知到异常。

但是，接下来内存表的这个特性就会让使用现象显得更“诡异”了。由于MySQL知道重启之后，内存表的数据会丢失。所以，担心主库重启之后，出现主备不一致，MySQL在实现上做了这样一

件事儿：在数据库重启之后，往binlog里面写入一行`DELETE FROM t1`。

如果你使用是如图9所示的双M结构的话：

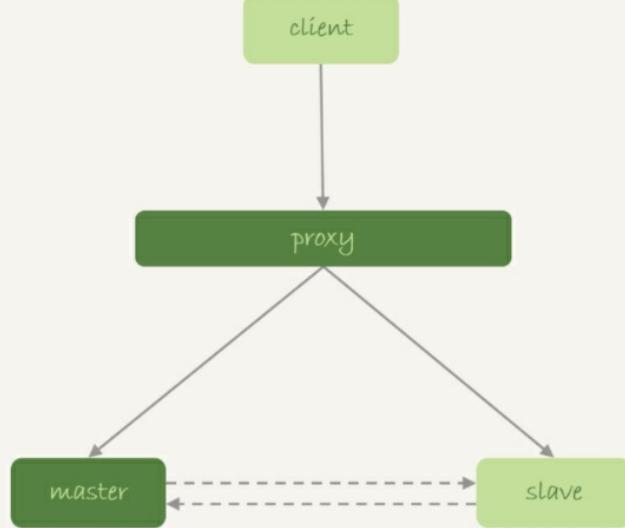


图9 双M结构

在备库重启的时候，备库binlog里的`delete`语句就会传到主库，然后把主库存表的内容删除。这样你在使用的时候就会发现，主库的内存表数据突然被清空了。

基于上面的分析，你可以看到，内存表并不适合在生产环境上作为普通数据表使用。

有同学会说，但是内存表执行速度快呀。这个问题，其实你可以这么分析：

1. 如果你的表更新量大，那么并发度是一个很重要的参考指标，InnoDB支持行锁，并发度比内存表好；
2. 能放到内存表的数据量都不大。如果你考虑的是读的性能，一个读QPS很高并且数据量不大的表，即使是使用InnoDB，数据也是都会缓存在InnoDB Buffer Pool里的。因此，使用InnoDB表的读性能也不会差。

所以，我建议你把普通内存表都用InnoDB表来代替。但是，有一个场景却是例外的。

这个场景就是，我们在第35和36篇说到的用户临时表。在数据量可控，不会耗费过多内存的情况下，你可以考虑使用内存表。

内存临时表刚好可以无视内存表的两个不足，主要是下面的三个原因：

1. 临时表不会被其他线程访问，没有并发性的问题；
2. 临时表重启后也是需要删除的，清空数据这个问题不存在；
3. 备库的临时表也不会影响主库的用户线程。

现在，我们回过头再看一下第35篇join语句优化的例子，当时我建议的是创建一个InnoDB临时表，使用的语句序列是：

```
create temporary table temp_t(id int primary key, a int, b int, index(b))engine=innodb;
insert into temp_t select * from t2 where b>=1 and b<=2000;
select * from t1 join temp_t on (t1.b=temp_t.b);
```

了解了内存表的特性，你就知道了，其实这里使用内存临时表的效果更好，原因有三个：

1. 相比于InnoDB表，使用内存表不需要写磁盘，往表temp_t的写数据的速度更快；
2. 索引b使用hash索引，查找的速度比B-Tree索引快；
3. 临时表数据只有2000行，占用的内存有限。

因此，你可以对[第35篇文章](#)的语句序列做一个改写，将临时表t1改成内存临时表，并且在字段b上创建一个hash索引。

```
create temporary table temp_t(id int primary key, a int, b int, index (b))engine=memory;
insert into temp_t select * from t2 where b>=1 and b<=2000;
select * from t1 join temp_t on (t1.b=temp_t.b);
```

```

mysql> create temporary table temp_t(id int primary key, a int, b int, index (b))engine=memory;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into temp_t select * from t2 where b>=1 and b<=2000;
Query OK, 2000 rows affected (0.88 sec)
Records: 2000  Duplicates: 0  Warnings: 0

+----+----+----+----+----+----+
| 995 |    6 | 995 | 995 | 995 | 995 |
| 996 |    5 | 996 | 996 | 996 | 996 |
| 997 |    4 | 997 | 997 | 997 | 997 |
| 998 |    3 | 998 | 998 | 998 | 998 |
| 999 |    2 | 999 | 999 | 999 | 999 |
| 1000|    1 | 1000| 1000| 1000| 1000|
+----+----+----+----+----+----+
1000 rows in set (0.00 sec)

```

图10 使用内存临时表的执行效果

可以看到，不论是导入数据的时间，还是执行join的时间，使用内存临时表的速度都比使用InnoDB临时表要更快一些。

小结

今天这篇文章，我从“要不要使用内存表”这个问题展开，和你介绍了Memory引擎的几个特性。

可以看到，由于重启会丢数据，如果一个备库重启，会导致主备同步线程停止；如果主库跟这个备库是双M架构，还可能导致主库的内存表数据被删掉。

因此，在生产上，我不建议你使用普通内存表。

如果你是DBA，可以在建表的审核系统中增加这类规则，要求业务改用InnoDB表。我们在文中也分析了，其实InnoDB表性能还不错，而且数据安全也有保障。而内存表由于不支持行锁，更新语句会阻塞查询，性能也未必就如想象中那么好。

基于内存表的特性，我们还分析了它的一个适用场景，就是内存临时表。内存表支持hash索引，这个特性利用起来，对复杂查询的加速效果还是很不错的。

最后，我给你留一个问题吧。

假设你刚刚接手的一个数据库上，真的发现了一个内存表。备库重启之后肯定是会导致备库的内存表数据被清空，进而导致主备同步停止。这时，最好的做法是将它修改成InnoDB引擎表。

假设当时的业务场景暂时不允许你修改引擎，你可以加上什么自动化逻辑，来避免主备同步停止呢？

你可以把你的思考和分析写在评论区，我会在下一篇文章的末尾跟你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

今天文章的正文内容，已经回答了我们上期的问题，这里就不再赘述了。

评论区留言点赞板：

@老杨同志、@poppy、@长杰 这三位同学给出了正确答案，春节期间还持续保持跟进学习，给你们点赞。

The image shows a promotional graphic for a MySQL course. At the top left is the '极客时间' logo. The main title 'MySQL 实战 45 讲' is displayed prominently in large, bold, dark font. Below it is a subtitle '从原理到实战，丁奇带你搞懂 MySQL' in a smaller, lighter font. To the right of the text is a portrait photo of the instructor, Ding Qi, a young man with glasses and a black shirt, with his arms crossed. On the left side of the image, there is a section with the name '林晓斌' (Lin Xiaobin) and the text '网名丁奇 前阿里资深技术专家'. At the bottom, there is a call-to-action: '新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有现金奖励。'.

精选留言

Long 老师新年好 :-)
刚好遇到一个问题。

2

本来准备更新到，一个查询是怎么运行的里面的，看到这篇更新文章，就写在这吧，希望老师帮忙解答。

关于这个系统memory引擎表：`information_schema.tables`

相关信息如下

- (1) Verison: MySQL 5.6.26
- (2) 数据量`table_schema = abc`的有接近4W的表，整个实例有接近10W的表。（默认innodb引擎）
- (3) `mysql.user`和`mysql.db`的数据量都是100-200的行数，MyISAM引擎。

(4) 默认事务隔离级别RC

在运行查询语句1的时候： `select * from information_schema.tables where table_schema = 'abc';`

状态一直是**check permission, opening tables**, 其他线程需要打开的表在**open tables**里面被刷掉的，会显示在**opening tables**，可能需要小几秒后基本恢复正常。

但是如果在运行查询语句2: `select count(1) from information_schema.tables where table_schema = 'abc';` 这个时候语句2本身在**profiling**看长期处于**check permission**状态，其他线程就会出现阻塞现象，大部分卡在了**opening tables**，小部分**closing tables**。我测试下了，当个表查询的时候**check permission**大概也就是0.0005s左右的时间，4W个表理论良好状态应该是几十秒的事情。

但是语句1可能需要5-10分钟，语句2需要5分钟。

3个问题，请老师抽空看下：

- (1) `information_schema.tables`的组成方式，是我每次查询的时候从数据字典以及**data**目录下的文件中实时去读的吗？
- (2) 语句1和语句2在运行的时候的过程分别是怎样的，特别是语句2。
- (3) 语句2为什么会出现大量阻塞其他事务，其他事务都卡在**opening tables**的状态。

PS: 最后根据**audit log**分析来看，语句实际上是MySQL的一个客户端Toad发起的，当使用Toad的**object explorer**的界面来查询表，或者设置**connection**的时候指定的**default schema**是大域的时候就会run这个语句： (`table_schema`改成了abc，其他都是原样)

`SELECT COUNT(1) FROM information_schema.tables WHERE table_schema = 'abc' AND table_type != 'VIEW';`

再次感谢！

2019-02-08



放

1

老师新年快乐！过年都不忘给我们传授知识！

2019-02-08

| 作者回复

新年快乐！

2019-02-08



于家鹏

1

新年好！

课后作业：在备库配置跳过该内存表的主从同步。

有一个问题一直困扰着我：SSD以及云主机的广泛运用，像Innodb这种使用WAL技术似乎并不能发挥最大性能（我的理解：基于SSD的WAL更多的只是起到队列一样削峰填谷的作用）。对于一些数据量不是特别大，但读写频繁的应用（比如点赞、积分），有没有更好的引擎推荐。

2019-02-08

作者回复

即使是SSD，顺序写也比随机写快些的。不过确实没有机械盘那么明显。

2019-02-08



长杰

0

内存表一般数据量不大，并且更新不频繁，可以写一个定时任务，定期检测内存表的数据，如果数据不空，就将它持久化到一个innodb同结构的表中，如果为空，就反向将数据写到内存表中，这些操作可设置为不写入binlog。

2019-02-09



往事随风，顺其自然

0

为什么memory引擎中数据按照数组单独存储，0索引对应的数据怎么放到数组的最后

2019-02-09

作者回复

这就是堆组织表的数据存放方式

2019-02-09



HuaMax

0

课后题。是不是可以加上创建表的操作，并且是innodb类型的？

2019-02-09



老杨同志

0

安装之前学的知识，把主库delete语句的gtid，设置到从库中，就可以跳过这条语句了吧。

但是主备不一致是不是也要处理一下，将主库的内存表数据备份一下。然后delete数据，重新插入。

等备件执行者两个语句后，主备应该都有数据了

2019-02-08

作者回复

题目里说的是“备库重启”哈

2019-02-09

39 | 自增主键为什么不是连续的？

2019-02-11 林晓斌



在[第4篇文章](#)中，我们提到过自增主键，由于自增主键可以让主键索引尽量地保持递增顺序插入，避免了页分裂，因此索引更紧凑。

之前我见过有的业务设计依赖于自增主键的连续性，也就是说，这个设计假设自增主键是连续的。但实际上，这样的假设是错的，因为自增主键不能保证连续递增。

今天这篇文章，我们就来说说这个问题，看看什么情况下自增主键会出现“空洞”？

为了便于说明，我们创建一个表t，其中id是自增主键字段、c是唯一索引。

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `c` (`c`)
) ENGINE=InnoDB;
```

自增值保存在哪儿？

在这个空表t里面执行`insert into t values(null, 1, 1);`插入一行数据，再执行`show create table`命

令，就可以看到如下图所示的结果：

```
mysql> show create table t\G
***** 1. row *****
      Table: t
Create Table: CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `c` (`c`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

图1 自动生成的AUTO_INCREMENT值

可以看到，表定义里面出现了一个**AUTO_INCREMENT=2**，表示下一次插入数据时，如果需要自动生成自增值，会生成**id=2**。

其实，这个输出结果容易引起这样的误解：自增值是保存在表结构定义里的。实际上，表的结构定义存放在后缀名为**.frm**的文件中，但是并不会保存自增值。

不同的引擎对于自增值的保存策略不同。

- **MyISAM**引擎的自增值保存在数据文件中。
- **InnoDB**引擎的自增值，其实是保存在了内存里，并且到了**MySQL 8.0**版本后，才有了“自增值持久化”的能力，也就是才实现了“如果发生重启，表的自增值可以恢复为**MySQL**重启前的值”，具体情况是：
 - 在**MySQL 5.7**及之前的版本，自增值保存在内存里，并没有持久化。每次重启后，第一次打开表的时候，都会去找自增值的最大值**max(id)**，然后将**max(id)+1**作为这个表当前的自增值。

举例来说，如果一个表当前数据行里最大的**id**是10，**AUTO_INCREMENT=11**。这时候，我们删除**id=10**的行，**AUTO_INCREMENT**还是11。但如果马上重启实例，重启后这个表的**AUTO_INCREMENT**就会变成10。

也就是说，**MySQL**重启可能会修改一个表的**AUTO_INCREMENT**的值。

- 在**MySQL 8.0**版本，将自增值的变更记录在了**redo log**中，重启的时候依靠**redo log**恢复重启之前的值。

理解了**MySQL**对自增值的保存策略以后，我们再看看自增值修改机制。

自增值修改机制

在**MySQL**里面，如果字段**id**被定义为**AUTO_INCREMENT**，在插入一行数据的时候，自增值的

行为如下：

1. 如果插入数据时**id**字段指定为0、**null**或未指定值，那么就把这个表当前的**AUTO_INCREMENT**值填到自增字段；
2. 如果插入数据时**id**字段指定了具体的值，就直接使用语句里指定的值。

根据要插入的值和当前自增值的大小关系，自增值的变更结果也会有所不同。假设，某次要插入的值是X，当前的自增值是Y。

1. 如果X<Y，那么这个表的自增值不变；
2. 如果X≥Y，就需要把当前自增值修改为新的自增值。

新的自增值生成算法是：从**auto_increment_offset**开始，以**auto_increment_increment**为步长，持续叠加，直到找到第一个大于X的值，作为新的自增值。

其中，**auto_increment_offset**和**auto_increment_increment**是两个系统参数，分别用来表示自增的初始值和步长，默认值都是1。

备注：在一些场景下，使用的就不全是默认值。比如，双M的主备结构里要求双写的时候，我们就可能会设置成**auto_increment_increment=2**，让一个库的自增**id**都是奇数，另一个库的自增**id**都是偶数，避免两个库生成的主键发生冲突。

当**auto_increment_offset**和**auto_increment_increment**都是1的时候，新的自增值生成逻辑很简单，就是：

1. 如果准备插入的值>=当前自增值，新的自增值就是“准备插入的值+1”；
2. 否则，自增值不变。

这就引入了我们文章开头提到的问题，在这两个参数都设置为1的时候，自增主键**id**却不能保证是连续的，这是什么原因呢？

自增值的修改时机

要回答这个问题，我们就要看一下自增值的修改时机。

假设，表t里面已经有了(1,1,1)这条记录，这时我再执行一条插入数据命令：

```
insert into t values(null, 1, 1);
```

这个语句的执行流程就是：

1. 执行器调用InnoDB引擎接口写入一行，传入的这一行的值是(0,1,1);
2. InnoDB发现用户没有指定自增id的值，获取表t当前的自增值2;
3. 将传入的行的值改成(2,1,1);
4. 将表的自增值改成3;
5. 继续执行插入数据操作，由于已经存在c=1的记录，所以报Duplicate key error，语句返回。

对应的执行流程图如下：

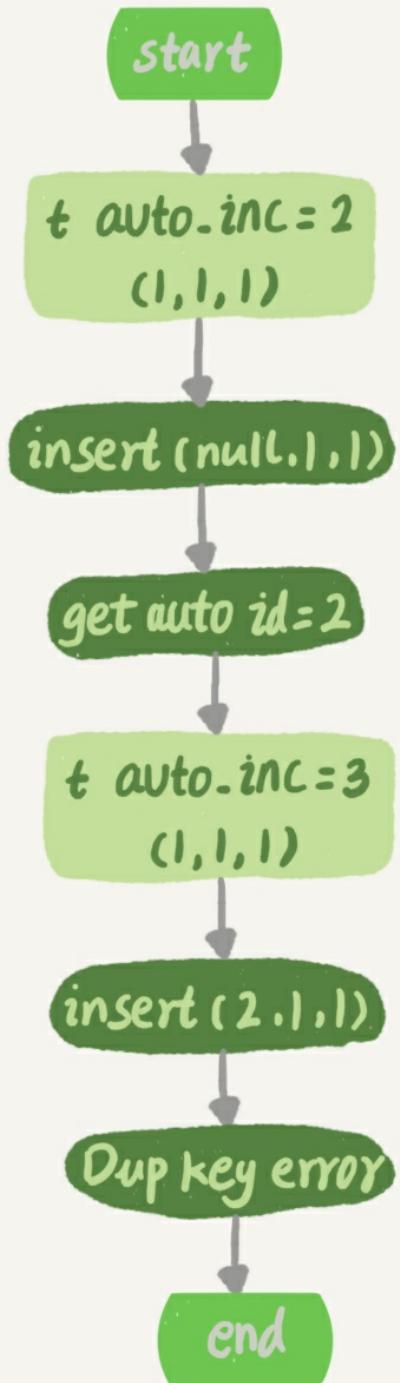


图2 `insert(null, 1,1)`唯一键冲突

可以看到，这个表的自增值改成3，是在真正执行插入数据的操作之前。这个语句真正执行的时候，因为碰到唯一键冲突，所以id=2这一行并没有插入成功，但也没有将自增值再改回去。

所以，在这之后，再插入新的数据行时，拿到的自增id就是3。也就是说，出现了自增主键不连续的情况。

如图3所示就是完整的演示结果。

```
mysql> CREATE TABLE `t` (
    ->   `id` int(11) NOT NULL AUTO_INCREMENT,
    ->   `c` int(11) DEFAULT NULL,
    ->   `d` int(11) DEFAULT NULL,
    ->   PRIMARY KEY (`id`),
    ->   UNIQUE KEY `c` (`c`)
    -> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t values(null,1,1);
Query OK, 1 row affected (0.00 sec)

mysql> insert into t values(null,1,1);
ERROR 1062 (23000): Duplicate entry '1' for key 'c'
mysql> insert into t values(null,2,2);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+----+----+----+
| id | c  | d  |
+----+----+----+
| 1  | 1  | 1  |
| 3  | 2  | 2  |
+----+----+----+
2 rows in set (0.00 sec)
```

图3一个自增主键id不连续的复现步骤

可以看到，这个操作序列复现了一个自增主键id不连续的现场(没有id=2的行)。可见，唯一键冲突是导致自增主键id不连续的第一种原因。

同样地，事务回滚也会产生类似的现象，这就是第二种原因。

下面这个语句序列就可以构造不连续的自增id，你可以自己验证一下。

```
insert into t values(null,1,1);
begin;
insert into t values(null,2,2);
rollback;
insert into t values(null,2,2);
//插入的行是(3,2,2)
```

你可能会问，为什么在出现唯一键冲突或者回滚的时候，MySQL没有把表t的自增值改回去呢？如果把表t的当前自增值从3改回2，再插入新数据的时候，不就可以生成id=2的一行数据了吗？

其实，MySQL这么设计是为了提升性能。接下来，我就跟你分析一下这个设计思路，看看自增值为什么不能回退。

假设有两个并行执行的事务，在申请自增值的时候，为了避免两个事务申请到相同的自增id，肯定要加锁，然后顺序申请。

1. 假设事务A申请到了id=2，事务B申请到id=3，那么这时候表t的自增值是4，之后继续执行。
2. 事务B正确提交了，但事务A出现了唯一键冲突。
3. 如果允许事务A把自增id回退，也就是把表t的当前自增值改回2，那么就会出现这样的情况：表里面已经有id=3的行，而当前的自增id值是2。
4. 接下来，继续执行的其他事务就会申请到id=2，然后再申请到id=3。这时，就会出现插入语句报错“主键冲突”。

而为了解决这个主键冲突，有两种方法：

1. 每次申请id之前，先判断表里面是否已经存在这个id。如果存在，就跳过这个id。但是，这个方法的成本很高。因为，本来申请id是一个很快的操作，现在还要再去主键索引树上判断id是否存在。
2. 把自增id的锁范围扩大，必须等到一个事务执行完成并提交，下一个事务才能再申请自增id。这个方法的问题，就是锁的粒度太大，系统并发能力大大下降。

可见，这两个方法都会导致性能问题。造成这些麻烦的罪魁祸首，就是我们假设的这个“允许自增id回退”的前提导致的。

因此，InnoDB放弃了这个设计，语句执行失败也不回退自增id。也正是因为这样，所以才只保证了自增id是递增的，但不保证是连续的。

自增锁的优化

可以看到，自增id锁并不是一个事务锁，而是每次申请完就马上释放，以便允许别的事务再申请。其实，在MySQL 5.1版本之前，并不是这样的。

接下来，我会先给你介绍下自增锁设计的历史，这样有助于你分析接下来的一个问题。

在MySQL 5.0版本的时候，自增锁的范围是语句级别。也就是说，如果一个语句申请了一个表自增锁，这个锁会等语句执行结束以后才释放。显然，这样设计会影响并发度。

MySQL 5.1.22版本引入了一个新策略，新增参数`innodb_autoinc_lock_mode`，默认值是1。

1. 这个参数的值被设置为0时，表示采用之前MySQL 5.0版本的策略，即语句执行结束后才释放锁；
2. 这个参数的值被设置为1时：
 - 普通`insert`语句，自增锁在申请之后就马上释放；
 - 类似`insert ...select`这样的批量插入数据的语句，自增锁还是要等语句结束后才被释放；
3. 这个参数的值被设置为2时，所有的申请自增主键的动作都是申请后就释放锁。

你一定有两个疑问：为什么默认设置下，`insert ...select`要使用语句级的锁？为什么这个参数的默认值不是2？

答案是，这么设计还是为了数据的一致性。

我们一起来看一下这个场景：

session A	session B
<code>insert into t values(null, 1,1);</code> <code>insert into t values(null, 2,2);</code> <code>insert into t values(null, 3,3);</code> <code>insert into t values(null, 4,4);</code>	
	<code>create table t2 like t;</code>
<code>insert into t2 values(null, 5,5);</code>	<code>insert into t2(c,d) select c,d from t;</code>

图4 批量插入数据的自增锁

在这个例子里，我往表t1中插入了4行数据，然后创建了一个相同结构的表t2，然后两个session同时执行向表t2中插入数据的操作。

你可以设想一下，如果session B是申请了自增值以后马上就释放自增锁，那么就可能出现这样的情况：

- session B先插入了两个记录，(1,1,1)、(2,2,2);
- 然后，session A来申请自增id得到id=3，插入了(3,5,5);
- 之后，session B继续执行，插入两条记录(4,3,3)、(5,4,4)。

你可能会说，这也没关系吧，毕竟session B的语义本身就没有要求表t2的所有行的数据都跟session A相同。

是的，从数据逻辑上看是对的。但是，如果我们现在的binlog_format=statement，你可以设想下，binlog会怎么记录呢？

由于两个session是同时执行插入数据命令的，所以binlog里面对表t2的更新日志只有两种情况：要么先记session A的，要么先记session B的。

但不论是哪一种，这个binlog拿去从库执行，或者用来恢复临时实例，备库和临时实例里面，session B这个语句执行出来，生成的结果里面，id都是连续的。这时，这个库就发生了数据不一致。

你可以分析一下，出现这个问题的原因是什么？

其实，这是因为原库session B的insert语句，生成的id不连续。这个不连续的id，用statement格式的binlog来串行执行，是执行不出来的。

而要解决这个问题，有两种思路：

1. 一种思路是，让原库的批量插入数据语句，固定生成连续的id值。所以，自增锁直到语句执行结束才释放，就是为了达到这个目的。
2. 另一种思路是，在binlog里面把插入数据的操作都如实记录进来，到备库执行的时候，不再依赖于自增主键去生成。这种情况，其实就是innodb_autoinc_lock_mode设置为2，同时binlog_format设置为row。

因此，在生产上，尤其是有insert ... select这种批量插入数据的场景时，从并发插入数据性能的角度考虑，我建议你这样设置：innodb_autoinc_lock_mode=2，并且binlog_format=row。这样做，既能提升并发性，又不会出现数据一致性问题。

需要注意的是，我这里说的批量插入数据，包含的语句类型是insert ... select、replace ... select和load data语句。

但是，在普通的insert语句里面包含多个value值的情况下，即使innodb_autoinc_lock_mode设置为1，也不会等语句执行完成才释放锁。因为这类语句在申请自增id的时候，是可以精确计算出需要多少个id的，然后一次性申请，申请完成后锁就可以释放了。

也就是说，批量插入数据的语句，之所以需要这么设置，是因为“不知道要预先申请多少个id”。

既然预先不知道要申请多少个自增id，那么一种直接的想法就是需要一个时申请一个。但如果一个select ...insert语句要插入10万行数据，按照这个逻辑的话就要申请10万次。显然，这种申请自增id的策略，在大批量插入数据的情况下，不但速度慢，还会影响并发插入的性能。

因此，对于批量插入数据的语句，MySQL有一个批量申请自增id的策略：

1. 语句执行过程中，第一次申请自增id，会分配1个；
2. 1个用完以后，这个语句第二次申请自增id，会分配2个；
3. 2个用完以后，还是这个语句，第三次申请自增id，会分配4个；
4. 依此类推，同一个语句去申请自增id，每次申请到的自增id个数都是上一次的两倍。

举个例子，我们一起看看下面的这个语句序列：

```
insert into t values(null, 1,1);
insert into t values(null, 2,2);
insert into t values(null, 3,3);
insert into t values(null, 4,4);
create table t2 like t;
insert into t2(c,d) select c,d from t;
insert into t2 values(null, 5,5);
```

insert..select，实际上往表t2中插入了4行数据。但是，这四行数据是分三次申请的自增id，第一次申请到了id=1，第二次被分配了id=2和id=3，第三次被分配到id=4到id=7。

由于这条语句实际只用上了4个id，所以id=5到id=7就被浪费掉了。之后，再执行insert into t2 values(null, 5,5)，实际上插入的数据就是(8,5,5)。

这是主键id出现自增id不连续的第三种原因。

小结

今天，我们从“自增主键为什么会出现不连续的值”这个问题开始，首先讨论了自增值的存储。

在MyISAM引擎里面，自增值是被写在数据文件上的。而在InnoDB中，自增值是被记录在内存的。MySQL直到8.0版本，才给InnoDB表的自增值加上了持久化的能力，确保重启前后一个表的自增值不变。

然后，我和你分享了一个语句执行过程中，自增值改变的时机，分析了为什么MySQL在事务回滚的时候不能回收自增id。

MySQL 5.1.22版本开始引入的参数`innodb_autoinc_lock_mode`, 控制了自增值申请时的锁范围。从并发性能的角度考虑, 我建议你将其设置为2, 同时将`binlog_format`设置为row。我在前面的文章中其实多次提到, `binlog_format`设置为row, 是很有必要的。今天的例子给这个结论多了一个理由。

最后, 我给你留一个思考题吧。

在最后一个例子中, 执行`insert into t2(c,d) select c,d from t;`这个语句的时候, 如果隔离级别是可重复读 (`repeatable read`), `binlog_format=statement`。这个语句会对表t的所有记录和间隙加锁。

你觉得为什么需要这么做呢?

你可以把你的思考和分析写在评论区, 我会在下一篇文章和你讨论这个问题。感谢你的收听, 也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是, 如果你维护的MySQL系统里有内存表, 怎么避免内存表突然丢数据, 然后导致主备同步停止的情况。

我们假设的是主库暂时不能修改引擎, 那么就把备库的内存表引擎先都改成InnoDB。对于每个内存表, 执行

```
set sql_log_bin=off;
alter table tbl_name engine=innodb;
```

这样就能避免备库重启的时候, 数据丢失的问题。

由于主库重启后, 会往binlog里面写“`delete from tbl_name`”, 这个命令传到备库, 备库的同名的表数据也会被清空。

因此, 就不会出现主备同步停止的问题。

如果由于主库异常重启, 触发了HA, 这时候我们之前修改过引擎的备库变成了主库。而原来的主库变成了新备库, 在新备库上把所有的内存表 (这时候表里没数据) 都改成InnoDB表。

所以, 如果我们不能直接修改主库上的表引擎, 可以配置一个自动巡检的工具, 在备库上发现内存表就把引擎改了。

同时, 跟业务开发同学约定好建表规则, 避免创建新的内存表。

评论区留言点赞板：

大家在春节期间还坚持看专栏，并且深入地思考和回复，给大家点赞。

@长杰 同学提到的将数据保存到InnoDB表用来持久化，也是一个方法。不过，我还是建议釜底抽薪，直接修改备库的内存表的引擎。

@老杨同志 提到的是主库异常重启的场景，这时候是不会报主备不一致的，因为主库重启的时候写了`delete from tbl_name`，主备的内存表都清空了。

The banner features the '极客时间' logo at the top left. The main title 'MySQL 实战 45 讲' is prominently displayed in large, bold, dark font. Below it, a subtitle reads '从原理到实战，丁奇带你搞懂 MySQL' in a smaller, lighter font. To the right of the text is a portrait photo of Ding Qi, a man with short dark hair and glasses, wearing a black button-down shirt, with his arms crossed. At the bottom left, there's a section for the author: '林晓斌' (Ling Xiaobin) with the note '网名丁奇 前阿里资深技术专家'. A call-to-action at the bottom says '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。' (New version upgrade: Click 'Ask friends to read', get 10 free reads, and invite to subscribe for cash rewards.)

精选留言



长杰

6

在最后一个例子中，执行 `insert into t2(c,d) select c,d from t;` 这个语句的时候，如果隔离级别是可重复读（repeatable read），`binlog_format=statement`。这个语句会对表 `t` 的所有记录和间隙加锁。

你觉得为什么需要这么做呢？

假如原库不对 `t` 表所有记录和间隙加锁，如果有其他事物新增数据并先与这个批量操作提交，由于事物的隔离级别是可重复读，`t2` 是看不到新增的数据的。但是记录的 `binlog` 是 `statement` 格式，备库或基于 `binlog` 恢复的临时库，`t2` 会看到新增的数据，出现数据不一致的情况。

2019-02-11

| 作者回复

[] 这是一个典型的场景

2019-02-11



aliang

1

老师，我们这边有的开发不喜欢用mysql自带的主键自增功能，而是在程序中控制主键（时间+业务+机器+序列，**bigint**类型，实际长度有17位，其中序列保存在内存中，每次递增，主键值不连续）。理由是

- (1) 通过这样的主键可以直接定位数据，减少索引
- (2) 如果自增，必须先存数据得到主键才可继续下面的程序，如果自己计算主键，可以在入库前进行异步处理
- (3) a表要insert得到主键，然后处理b表，然后根据条件还要update a表。如果程序自己控制，就不用先insert a表，数据可以在内存中，直到最后一次提交。（对于a表，本来是insert+update，最后只是一条insert，少一次数据库操作）

我想请问的是：

- (1) 针对理由1，是否可以用组合索引替代？
- (2) 针对理由2，是否mysql自身的主键自增分配逻辑就已经能实现了？
- (3) 针对理由3，主键更长意味着更大的索引（主键索引和普通索引），你觉得怎样做会更好呢

2019-02-12

| 作者回复

“（时间+业务+机器+序列，**bigint**类型，实际长度有17位，其中序列保存在内存中，每次递增，主键值不连续）。”----**bigint**就是8位，这个你需要确定一下。如果是8位的还好，如果是17位的字符串，就比较耗费空间；

- (1) 如果“序列”是递增的，还是不能直接用来体现业务逻辑吧？创建有业务意义的字段索引估计还是省不了的？
- (2) mysql确实做不到“插入之前就先算好接下来的id是多少”，一般都是insert执行完成后，再执行select last_insert_id
- (3) 先insert a再update b再update a，确实看上去比较奇怪，不过感觉这个逻辑应该是可以优化的，不应该作为“主键选择”的一个依据。你可否脱敏一下，把模拟的表结构和业务逻辑说下，看看是不是可以优化的。

总之，按照你说的“时间+业务+机器+序列”这种模式，有点像用**uuid**，主要的问题还是，如果这个表的索引多，占用的空间比较大

2019-02-12



进阶的码农

0

上期问题解答，有点疑问

```
set sql_log_bin=off;  
alter table tbl_name engine=innodb;
```

为什么备库需要执行set sql_log_bin=off这一句

把表的引擎改成innodb不就能解决重启后内存表被删除的问题吗？

2019-03-12



进阶的码农

0



课后题

在最后一个例子中，执行 `insert into t2(c,d) select c,d from t;` 这个语句的时候，如果隔离级别是可重复读（repeatable read），`binlog_format=statement`会加记录锁和间隙锁。啥我的`binlog_format=row`也加锁了

2019-03-12



hetiu

0

老师，请问下`innodb_autoinc_lock_mode`配置是库级别的还是实例级别的？

2019-03-05

| 作者回复

全局的

2019-03-06



二十四桥仍在

0

UUID生成主键

2019-03-05



唐名之

0

老师，如果我业务场景必须需要一个带有序自增值，设业务为表A，另外添加一张表记录自增为表B，表B包含3个字段（自增主键，表A唯一键，自增列）；伪代码如下；这样能实现吗？或者有其他什么好的方案？

```
begin;  
insert into A values (字段1, 唯一键) ;  
insert into B value (表A唯一键, 自增列);  
commit;
```

2019-02-25

| 作者回复

这样思路上是ok的，

不过表B怎么有两个自增列？一个表只能有一个自增列。

2019-02-25



AstonPutting

0

老师，`innodb_autoinc_lock_mode = 2, binlog_format = statement` 不也会出现数据不一致的问题吗？不是很理解 `binlog_format = statement` 的情况下，1与2的区别。

2019-02-21

| 作者回复

`innodb_autoinc_lock_mode = 2`的时候就要`binlog_format = row`才好

2019-02-21



Ryoma

0

在8.0.3版本后，`innodb_autoinc_lock_mode`默认值已是2，在`binlog_format`默认值为row的前提下，想来也是为了增加并发。

https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_autoinc_lock_mode

2019-02-14

| 作者回复

|| 大势所趋 ||

2019-02-16



帽子掉了

0

老师您好，我有一个时序问题，想请教一下。

从这篇文章的介绍来看，获取自增id和写binlog是有先后顺序的。

那么在binlog为statement的情况下。

语句A先获取id=1，然后B获取id=2，接着B提交，写binlog，再A写binlog。

这个时候如果binlog重放，是不是会发生B的id为1，而A的id为2的不一致的情况？

2019-02-13

| 作者回复

好问题，不会

因为binlog在记录这种带自增值的语句之前，会在前面多一句，用于指定“接下来这个语句需要的自增ID值是多少”，而这个值，是在主库上这一行插入成功后对应的自增值，所以是一致的

2019-02-14



郭烊千玺

0

请教老师个额外话题 select concat(truncate(sum(data_length)/1024/1024,2),'MB') as data_size, concat(truncate(sum(max_data_length)/1024/1024,2),'MB') as max_data_size, concat(truncate(sum(data_free)/1024/1024,2),'MB') as data_free, concat(truncate(sum(index_length)/1024/1024,2),'MB') as index_size from information_schema.tables where TABLE_SCHEMA = 'databasename'; 网上广为流传的这个统计的表大小的方法准确吗 mysql内部是怎么统计的？并且data_free这个mydql内部又是怎么统计的是采样8个页来评估整表吗 并且实验总感觉这样统计不准啊 到底靠谱吗 求赐教求赐教啊 困惑好久了

2019-02-12



悟空

0

赶上了进度，把春节期间的补回来了

2019-02-12

| 作者回复

||

2019-02-12



we

0

```
insert into t values(null,1,1);
begin;
insert into t values(null,2,2);
```

```
rollback;  
insert into t values(null,2,2);  
// 插入的行是 (3,2,2)
```

老师 里面是 rollback 吧

2019-02-12

| 作者回复

是的，我手残了。。。

多谢指出，发起勘误了哈

2019-02-12



牛在天上飞

0

老师，请问产生大量的event事件会对mysql服务器有什么影响？主要是哪几个方面的影响？

2019-02-12

| 作者回复

也没啥，主要就是不好管理。。

毕竟event是写在MySQL里的，写程序的同学不一定会记得。

比较建议将这类逻辑写在应用程序里面

2019-02-12



aliang

0

老师，执行SELECT `ID`, `USER`, `HOST`, `DB`, `COMMAND`, `TIME`, `STATE`, LEFT(`INFO`, 51200) AS `Info` FROM `information_schema`.`PROCESSLIST`;后不时有COMMAND为killed但info为null的进程，请问是怎么回事呢

2019-02-11

| 作者回复

就表示还在“killed”状态，看一下32篇哈

2019-02-11



陈华应

0

防止insert语句执行过程中，原表有新增数据，进而导致的插入新表的数据比原表少

2019-02-11

| 作者回复

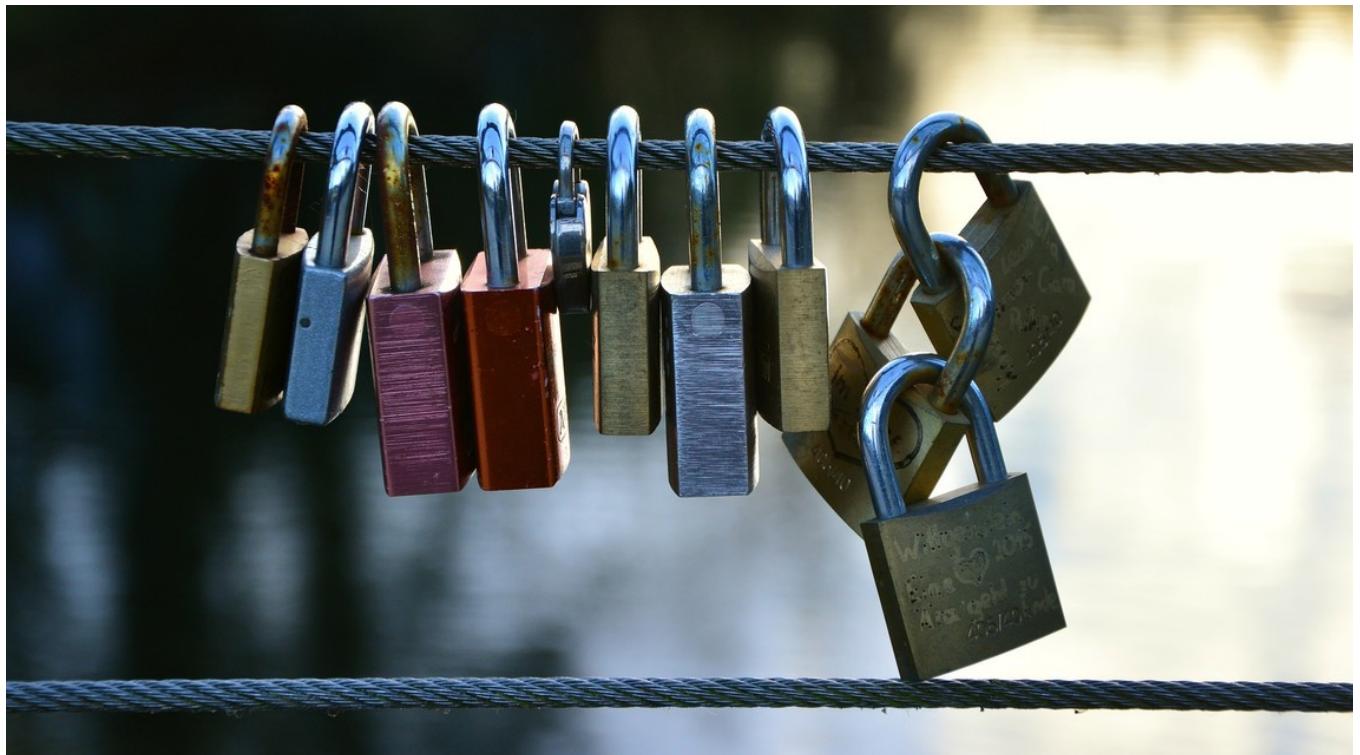
确实是考虑并发，不过并不会有这个现象哦，因为一个语句执行期间还是有一致性视图的。

把binlog加进去考虑下哈

2019-02-11

40 | insert语句的锁为什么这么多？

2019-02-13 林晓斌



在上一篇文章中，我提到MySQL对自增主键锁做了优化，尽量在申请到自增id以后，就释放自增锁。

因此，`insert`语句是一个很轻量的操作。不过，这个结论对于“普通的`insert`语句”才有效。也就是说，还有些`insert`语句是属于“特殊情况”的，在执行过程中需要给其他资源加锁，或者无法在申请到自增id以后就立马释放自增锁。

那么，今天这篇文章，我们就一起来聊聊这个话题。

`insert ... select` 语句

我们先从昨天的问题说起吧。表t和t2的表结构、初始化数据语句如下，今天的例子我们还是针对这两个表展开。

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `c` (`c`)
) ENGINE=InnoDB;
```

```
insert into t values(null, 1,1);
insert into t values(null, 2,2);
insert into t values(null, 3,3);
insert into t values(null, 4,4);
```

```
create table t2 like t
```

现在，我们一起来看看为什么在可重复读隔离级别下，`binlog_format=statement`时执行：

```
insert into t2(c,d) select c,d from t;
```

这个语句时，需要对表t的所有行和间隙加锁呢？

其实，这个问题我们需要考虑的还是日志和数据的一致性。我们看下这个执行序列：

session A	session B
insert into t values(-1, -1,-1);	insert into t2(c,d) select c,d from t;

图1 并发insert场景

实际的执行效果是，如果**session B**先执行，由于这个语句对表t主键索引加了 $(-\infty, 1]$ 这个**next-key lock**，会在语句执行完成后，才允许**session A**的**insert**语句执行。

但如果没锁的话，就可能出现**session B**的**insert**语句先执行，但是后写入**binlog**的情况。于是，在`binlog_format=statement`的情况下，**binlog**里面就记录了这样的语句序列：

```
insert into t values(-1,-1,-1);
insert into t2(c,d) select c,d from t;
```

这个语句到了备库执行，就会把id=1这一行也写到表t2中，出现主备不一致。

insert 循环写入

当然了，执行`insert ...select` 的时候，对目标表也不是锁全表，而是只锁住需要访问的资源。

如果现在有这么一个需求：要往表t2中插入一行数据，这一行的c值是表t中c值的最大值加1。

此时，我们可以这么写这条SQL语句：

```
insert into t2(c,d) (select c+1, d from t force index(c) order by c desc limit 1);
```

这个语句的加锁范围，就是表t索引c上的(3,4]和(4,supremum]这两个next-key lock，以及主键索引上id=4这一行。

它的执行流程也比较简单，从表t中按照索引c倒序，扫描第一行，拿到结果写入到表t2中。

因此整条语句的扫描行数是1。

这个语句执行的慢查询日志（slow log），如下图所示：

```
# Query_time: 0.000732 Lock_time: 0.000356 Rows_sent: 0 Rows_examined: 1
SET timestamp=1548852517;
insert into t2(c,d) (select c+1, d from t force index(c) order by c desc limit 1);
```

图2 慢查询日志—将数据插入表t2

通过这个慢查询日志，我们看到`Rows_examined=1`，正好验证了执行这条语句的扫描行数为1。

那么，如果我们是要把这样的一行数据插入到表t中的话：

```
insert into t(c,d) (select c+1, d from t force index(c) order by c desc limit 1);
```

语句的执行流程是怎样的？扫描行数又是多少呢？

这时候，我们再看慢查询日志就会发现不对了。

```
# Query_time: 0.000478 Lock_time: 0.000128 Rows_sent: 0 Rows_examined: 5
SET timestamp=1548852287;
insert into t(c,d) (select c+1, d from t force index(c) order by c desc limit 1);
```

图3 慢查询日志—将数据插入表t

可以看到，这时候的`Rows_examined`的值是5。

我在前面的文章中提到过，希望你都能够学会用explain的结果来“脑补”整条语句的执行过程。今天，我们就来一起试试。

如图4所示就是这条语句的explain结果。

mysql> explain insert into t(c,d) (select c+1, d from t force index(c) order by c desc limit 1);											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	INSERT	t	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL	Using temporary
1	SIMPLE	t	NULL	index	NULL	c	5	NULL	1	100.00	

图4 explain结果

从Extra字段可以看到“Using temporary”字样，表示这个语句用到了临时表。也就是说，执行过程中，需要把表t的内容读出来，写入临时表。

图中rows显示的是1，我们不妨先对这个语句的执行流程做一个猜测：如果说的是把子查询的结果读出来（扫描1行），写入临时表，然后再从临时表读出来（扫描1行），写回表t中。那么，这个语句的扫描行数就应该是2，而不是5。

所以，这个猜测不对。实际上，Explain结果里的rows=1是因为受到了limit 1的影响。

从另一个角度考虑的话，我们可以看看InnoDB扫描了多少行。如图5所示，是在执行这个语句前后查看Innodb_rows_read的结果。

mysql> show status like '%Innodb_rows_read%';	
Variable_name	Value
Innodb_rows_read	2242
1 row in set (0.00 sec)	
mysql> insert into t(c,d) (select c+1, d from t force index(c) order by c desc limit 1);	Query OK, 1 row affected (0.00 sec)
Records: 1 Duplicates: 0 Warnings: 0	
mysql> show status like '%Innodb_rows_read%';	
Variable_name	Value
Innodb_rows_read	2246
1 row in set (0.00 sec)	

图5 查看 Innodb_rows_read变化

可以看到，这个语句执行前后，Innodb_rows_read的值增加了4。因为默认临时表是使用Memory引擎的，所以这4行查的都是表t，也就是说对表t做了全表扫描。

这样，我们就把整个执行过程理清楚了：

1. 创建临时表，表里有两个字段c和d。

2. 按照索引c扫描表t，依次取c=4、3、2、1，然后回表，读到c和d的值写入临时表。这时，Rows_examined=4。
 3. 由于语义里面有limit 1，所以只取了临时表的第一行，再插入到表t中。这时，Rows_examined的值加1，变成了5。
- 也就是说，这个语句会导致在表t上做全表扫描，并且会给索引c上的所有间隙都加上共享的next-key lock。所以，这个语句执行期间，其他事务不能在这个表上插入数据。
- 至于这个语句的执行为什么需要临时表，原因是这类一边遍历数据，一边更新数据的情况，如果读出来的数据直接写回原表，就可能在遍历过程中，读到刚刚插入的记录，新插入的记录如果参与计算逻辑，就跟语义不符。
- 由于实现上这个语句没有在子查询中就直接使用limit 1，从而导致了这个语句的执行需要遍历整个表t。它的优化方法也比较简单，就是用前面介绍的方法，先insert into到临时表temp_t，这样就只需要扫描一行；然后再从表temp_t里面取出这行数据插入表t1。
- 当然，由于这个语句涉及的数据量很小，你可以考虑使用内存临时表来做这个优化。使用内存临时表优化时，语句序列的写法如下：

```
create temporary table temp_t(c int,d int) engine=memory;
insert into temp_t (select c+1, d from t force index(c) order by c desc limit 1);
insert into t select * from temp_t;
drop table temp_t;
```

insert 唯一键冲突

前面的两个例子是使用insert ...select的情况，接下来我要介绍的这个例子就是最常见的insert语句出现唯一键冲突的情况。

对于有唯一键的表，插入数据时出现唯一键冲突也是常见的情况了。我先给你举一个简单的唯一键冲突的例子。

session A	session B
insert into t values(10,10,10);	
begin; insert into t values(11,10,10); (Duplicate entry '10' for key 'c')	
	insert into t values(12,9,9); (blocked)

图6 唯一键冲突加锁

这个例子也是在可重复读（repeatable read）隔离级别下执行的。可以看到，**session B**要执行的**insert**语句进入了锁等待状态。

也就是说，**session A**执行的**insert**语句，发生唯一键冲突的时候，并不只是简单地报错返回，还在冲突的索引上加了锁。我们前面说过，一个**next-key lock**就是由它右边界值定义的。这时候，**session A**持有索引c上的(5,10]共享**next-key lock**（读锁）。

至于为什么要加这个读锁，其实我也没有找到合理的解释。从作用上来看，这样做可以避免这一行被别的事务删掉。

这里[官方文档](#)有一个描述错误，认为如果冲突的是主键索引，就加记录锁，唯一索引才加**next-key lock**。但实际上，这两类索引冲突加的都是**next-key lock**。

备注：这个bug，是我在写这篇文章查阅文档时发现的，已经[发给官方](#)并被verified了。

有同学在前面文章的评论区问到，在有多个唯一索引的表中并发插入数据时，会出现死锁。但是，由于他没有提供复现方法或者现场，我也无法做分析。所以，我建议你在评论区发问题的时候，尽量同时附上复现方法，或者现场信息，这样我才好和你一起分析问题。

这里，我就先和你分享一个经典的死锁场景，如果你还遇到过其他唯一键冲突导致的死锁场景，也欢迎给我留言。

	session A	session B	session C
T1	begin; insert into t values(null, 5,5);		
T2		insert into t values(null, 5,5);	insert into t values(null, 5,5);
T3	rollback;		(Deadlock found)

图7 唯一键冲突—死锁

在**session A**执行**rollback**语句回滚的时候，**session C**几乎同时发现死锁并返回。

这个死锁产生的逻辑是这样的：

1. 在T1时刻，启动**session A**，并执行**insert**语句，此时在索引c的c=5上加了记录锁。注意，这个索引是唯一索引，因此退化为记录锁（如果你的印象模糊了，可以回顾下[第21篇文章](#)介绍的加锁规则）。

- 在T2时刻，session B要执行相同的insert语句，发现了唯一键冲突，加上读锁；同样地，session C也在索引c上，c=5这一个记录上，加了读锁。
- T3时刻，session A回滚。这时候，session B和session C都试图继续执行插入操作，都要加上写锁。两个session都要等待对方的行锁，所以就出现了死锁。

这个流程的状态变化图如下所示。

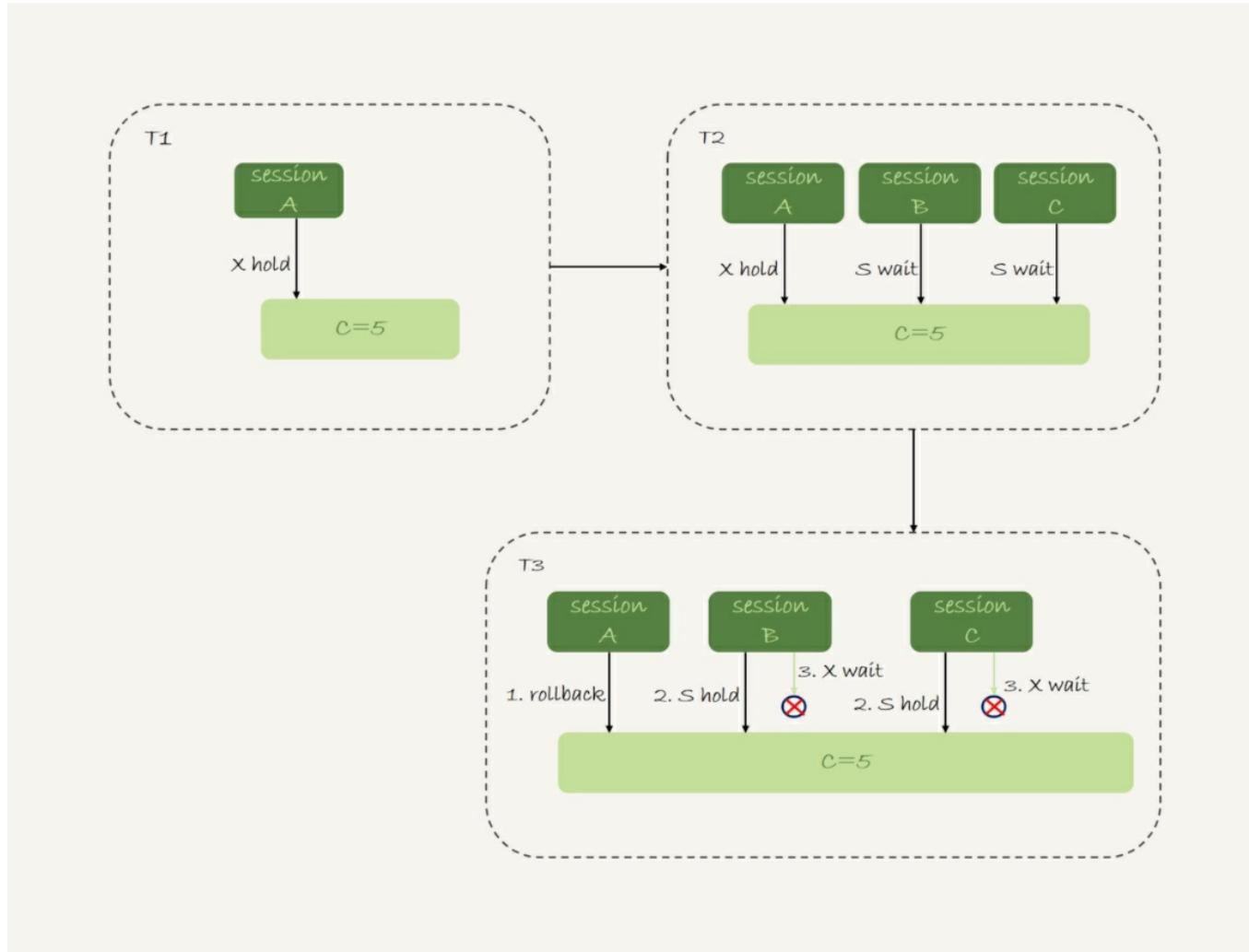


图8 状态变化图—死锁

insert into ... on duplicate key update

上面这个例子是主键冲突后直接报错，如果是改写成

```
insert into t values(11,10,10) on duplicate key update d=100;
```

的话，就会给索引c上[5,10]加一个排他的next-key lock（写锁）。

insert into ...on duplicate key update 这个语义的逻辑是，插入一行数据，如果碰到唯一键约束，就执行后面的更新语句。

注意，如果有多个列违反了唯一性约束，就会按照索引的顺序，修改跟第一个索引冲突的行。

现在表t里面已经有了(1,1,1)和(2,2,2)这两行，我们再来看看下面这个语句执行的效果：

```
mysql> insert into t values(2,1, 100) on duplicate key update d=100;
Query OK, 2 rows affected (0.00 sec)
mysql> select * from t where id<=2;
+---+---+---+
| id | c   | d   |
+---+---+---+
| 1  | 1   | 1   |
| 2  | 2   | 100 |
+---+---+---+
2 rows in set (0.00 sec)
```

图9 两个唯一键同时冲突

可以看到，主键id是先判断的，MySQL认为这个语句跟id=2这一行冲突，所以修改的是id=2的行。

需要注意的是，执行这条语句的affected rows返回的是2，很容易造成误解。实际上，真正更新的只有一行，只是在代码实现上，insert和update都认为自己成功了，update计数加了1，insert计数也加了1。

小结

今天这篇文章，我和你介绍了几种特殊情况下的insert语句。

insert ...select是很常见的在两个表之间拷贝数据的方法。你需要注意，在可重复读隔离级别下，这个语句会给select的表里扫描到的记录和间隙加读锁。

而如果insert和select的对象是同一个表，则有可能会造成循环写入。这种情况下，我们需要引入用户临时表来做优化。

insert语句如果出现唯一键冲突，会在冲突的唯一值上加共享的next-key lock(S锁)。因此，碰到由于唯一键约束导致报错后，要尽快提交或回滚事务，避免加锁时间过长。

最后，我给你留一个问题吧。

你平时在两个表之间拷贝数据用的是什么方法，有什么注意事项吗？在你的应用场景里，这个方法，相较于其他方法的优势是什么呢？

你可以把你的经验和分析写在评论区，我会在下一篇文章的末尾选取有趣的评论来和你一起分析。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期间题时间

我们已经在文章中回答了上期问题。

有同学提到，如果在`insert ...select`执行期间有其他线程操作原表，会导致逻辑错误。其实，这是不会的，如果不加锁，就是快照读。

一条语句执行期间，它的一致性视图是不会修改的，所以即使有其他事务修改了原表的数据，也不会影响这条语句看到的数据。

评论区留言点赞板：

@长杰 同学回答得非常准确。

The image is a promotional graphic for a MySQL course. It features a portrait of a man with glasses and a black shirt, standing with his arms crossed. To his left is the title 'MySQL 实战 45讲' and a subtitle '从原理到实战，丁奇带你搞懂 MySQL'.

林晓斌 网名丁奇
前阿里资深技术专家

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

精选留言



huolang

8

老师，死锁的例子，关于sessionA拿到的c=5的记录锁，sessionB和sessionC发现唯一键冲突会加上读锁我有几个疑惑：

1. sessionA拿到的c=5的记录锁是写锁吗？
2. 为什么sessionB和sessionC发现唯一键冲突会加上读锁？
3. 如果sessionA拿到c=5的记录所是写锁，那为什么sessionB和sessionC还能加c=5的读锁，写锁和读锁不应该是互斥的吗？
4. sessionA还没有提交，为什么sessionB和sessionC能发现唯一键冲突？

2019-02-13

作者回复

1. 是的
2. 这个我觉得是为了防止这个记录再被删除（不过这个理由不是很硬，我还没有找到其他解释
3. 互斥的，所以这两个语句都在等待。注意**next-key lock**是由间隙锁和记录锁组成的哦， 间隙锁加成功了的。好问题。
4. 还没有提交，但是这个记录已经作为最新记录写进去了，复习一下08篇哈

2019-02-14



老杨同志

4

课后问题：

我用的最多还是**insert into select**。如果数量比较大，会加上**limit 100,000**这种。并且看看后面的**select**条件是否走索引。缺点是会锁**select**的表。方法二：导出成**excel**，然后拼**sql**成**insert into values(),(),()**的形式。方法3，写类似淘宝调动的定时任务，任务的逻辑是查询**100**条记录，然后多个线程分到几个任务执行，比如是个线程，每个线程**10**条记录，插入后，在查询新的**10**条记录处理。

2019-02-13

作者回复

0

2019-02-14



sonic

3

你好，

我想问下文章中关于为什么需要创建临时表有这一句话：

如果读出来的数据直接写回原表，就可能在遍历过程中，读到刚刚插入的记录，新插入的记录如果参与计算逻辑，就跟语义不符。

我的疑问是：既然隔离级别是可重复读，照理来说新插入的记录应该不会参与计算逻辑呀。

2019-02-14

作者回复

可重复读隔离级别下，事务是可以看到自己刚刚修改的数据的，好问题

2019-02-16



滔滔

2

老师，之前提到的一个有趣的问题"A、B两个用户，如果互相喜欢，则成为好友。设计上是有两张表，一个是**like**表，一个是**friend**表，**like**表有**user_id**、**liker_id**两个字段，我设置为复合唯一索引即**uk_user_id_liker_id**。语句执行顺序是这样的：

以A喜欢B为例：

1、先查询对方有没有喜欢自己（B有没有喜欢A）

`select * from like where user_id = B and liker_id = A`

2、如果有，则成为好友

`insert into friend`

3、没有，则只是喜欢关系

`insert into like`，这个问题中如果把**select**语句改成“当前读”，则当出现A,B两个人同时喜欢对方

的情况下，是不是会出现由于"当前读"加的gap锁导致后面insert语句阻塞，从而发生死锁？

2019-02-13

作者回复

好问题

这种情况下一般是造成锁等待，不会造成死锁吧！

2019-02-14



夹心面包

2

1 关于insert造成死锁的情况,我之前做过测试,事务1并非只有insert,delete和update都可能造成死锁问题,核心还是插入唯一值冲突导致的.我们线上的处理办法是 1 去掉唯一值检测 2减少重复值的插入 3降低并发线程数量

2 关于数据拷贝大表我建议采用pt-archiver,这个工具能自动控制频率和速度,效果很不错,提议在低高峰期进行数据操作

2019-02-13

作者回复

[], 这两点都是很有用的建议

2019-02-13



王伯轩

1

老师你好,去年双11碰到了dbcrash掉的情况.至今没有找到答案,心里渗得慌.老师帮忙分析下. 我是一个开发,关于db的知识更多是在应用和基本原理上面,实在是找不到原因. 我也搜了一些资料 感觉像是mysql的bug,不过在其buglist中没有找到完全一致的, 当然也可能是我们业务也许导致库的压力大的原因.

应用端看到的现象是db没有响应, 应用需要访问db的线程全部僵死.db表现是hang住, 当时的诊断日志如下, 表面表现为一直获取不到latch锁(被一个insert线程持有不释放) <https://note.youdao.com/ynoteshare1/index.html?id=1771445db3ff1e08cbdd8328ea6765a7&type=note#/> 隔离级别是rr

同样的crash双11当天后面又出现了一次(哭死),

都是重启数据库解决的,

后面应用层面做了一样优化,没有再crash过, 优化主要如下:

1.减小读压力, 去除一些不必要的查询,

2.优化前, 有并发事务写和查询同一条数据记录, 即事务a执行insert尚未提交, 事务b就来查询(快照读), 优化后保证查询时insert事务已经提交

2019-02-19

作者回复

这就是压力太大了。. 一般伴随着ioutil很大, 语句执行特别慢, 别的语句就被堵着等锁, 等超时就自己crash

2019-02-19



nhanzhena-好客旅游网

1



循环插入数据，然后拿着刚刚插入的主键id，更新数据。请问怎么提高这个情况的效率

2019-02-15

| 作者回复

insert以后

select last_insert_id;

再update，

只能这么做啦

如果要快一些，可能可以考虑减少交互，比如写成存储过程

2019-02-16



伟仔_Hoo

0

老师，看到您的回复，当select c+1, d from t force index(c) order by c desc limit 1;这条语句单独执行是会在c索引上加[4,sup]这个next key lock，于是我进行了尝试

sessionA:

begin;

select c+1, d from t3 force index(c) order by c desc limit 1;

sessionB:

insert into t3 values(5, 5, 5);

结果是，sessionB插入成功，是不是我哪里理解错了？我的版本是5.7.23

2019-03-15

| 作者回复

session A的select语句没有加 for update 或者 lock in share mode ?

2019-03-16



猫小妖的尾巴

0

老师，我们的业务中有用到insert ...on duplicate key update导致死锁的情况，表是有唯一索引，DBA那边的解释是有唯一索引的insert需要两把锁，事务1先申请X锁成功，然后申请S锁，但是事务2正在申请X锁，与事务1的S锁冲突，系统决定回滚事务2，然后我就改成先查询存在直接update不存在再用原来的逻辑，不过我感觉还是不太明白，你可以解释一下吗

2019-03-10



涵涵妈 lillian

0

老师，能帮忙看下这个死锁记录吗？对于duplicate key插入有什么阻止的好方法？**LATEST DETECTED DEADLOCK**

190222 8:37:45

*** (1) TRANSACTION:

TRANSACTION 16FEC1AE, ACTIVE 0 sec inserting

mysql tables in use 1, locked 1

LOCK WAIT 6 lock struct(s), heap size 1248, 3 row lock(s)

MySQL thread id 169973, OS thread handle 0x2ba0fa040700, query id 41915315 10.45.133.181

W59FFHKU

INSERT INTO resource (

Id

, Name

, Date

, User

) VALUES (99127, 'RS_2098185e367d11e9878202a98a7af318', '', 'JR')

*** (1) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 78 page no 71 n bits 160 index `PRIMARY` of table `resource` trx id 16FEC1AE lock_mode X insert intention waiting

Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0

0: len 8; hex 73757072656d756d; asc supremum;;

*** (2) TRANSACTION:

TRANSACTION 16FEC1AF, ACTIVE 0 sec inserting

mysql tables in use 1, locked 1

6 lock struct(s), heap size 1248, 3 row lock(s)

MySQL thread id 169996, OS thread handle 0x2ba0ffec2700, query id 41915317 10.45.133.181

W59FFHKU

INSERT INTO resource (

Id

, Name

, Date

, User

) VALUES (99125, 'RS_2098b778367d11e9878202a98a7af318', '', 'JR')

*** (2) HOLDS THE LOCK(S):

RECORD LOCKS space id 78 page no 71 n bits 160 index `PRIMARY` of table `resource` trx id 16FEC1AF lock mode S

Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0

0: len 8; hex 73757072656d756d; asc supremum;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 78 page no 71 n bits 160 index `PRIMARY` of table `resource` trx id 16FEC1AF lock_mode X insert intention waiting

Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0

0: len 8; hex 73757072656d756d; asc supremum;;

*** WE ROLL BACK TRANSACTION (2)

2019-03-10



涵涵妈 lillian

0

老师，重复主键插入冲突是否推荐insert ignore方法？

2019-03-09

| 作者回复

这个取决于业务需求，如果是明确会存在这样的情况，并且可以忽略，是可以这么用的

2019-03-09



轻松的鱼

0

老师好，想请教一下死锁的例子中：

1. 在 session A rollback 前，session B/C 都因为唯一性冲突申请了 S Next-key lock，但是被 session A 的 X but not gap lock 阻塞；
2. 在 session A rollback 后，session B/C 顺利获得 S Next-key lock，并且都要继续进行插入，这时候我认为是因为插入意向锁（LOCK_INSERT_INTENTION）导致的死锁，因为插入意向锁会被 gap lock 阻塞，造成了相互等待。还没有进入到记录 X lock。
不知道我分析的对不对？

2019-03-06



张永志

0

对主键插入加读锁的个人理解，两个会话insert同样记录，在没有提交情况下，insert主键加读锁是为了避免第一个会话回滚后，第二个会话可以正常执行；第一个会话提交后，第二个会话再报错。

2019-02-28

| 作者回复

是为了实现这个目的，是吧？

2019-02-28



Mr.Strive.Z.H.L

0

老师您好：

关于文中的锁描述有所疑惑。

文中出现过 共享的next-key锁 和 排他的next-key锁。

我们知道next-key是由 gap lock 和 行锁组成的。

我一直以来的认知是 gap lock都是s锁，没有x锁。

而行锁有s锁和x锁。

比如 select.....lock in share mode， 行锁是s
锁。

比如select.....for update， 行锁就是x锁。

但是gap lock 始终是s锁。

文中直接描述next-key lock是排他的，总让我认为gap lock和行锁都是x锁。

不知道我理解得对不对？

2019-02-27

| 作者回复

是这样的，gap lock是无所谓S还是X的。

但是record lock 有。

Gap lock + 排他的record 就称作 排他的next-key lock 吧

2019-02-27



滔滔

0

老师，`select c+1, d from t force index(c) order by c desc limit 1;`这条语句如果单独执行，是会对表t进行全表加锁，还是只加`(3,4],(4,sup]`这两个next key锁。还有一个问题，这里为什么要加`force index(c)`，不加会是怎样的效果呢？

2019-02-24

| 作者回复

`(4,sup]`

以防优化器不走索引，影响我们结论（比如数据量比较小的时候）

2019-02-25



发条橙子。

0

老师，年后过来狂补课程了哈哈，看到老师的bug留言已经被fix掉准备在最新版本发布了呢。

这里我有一个疑问，我之前以为只有更新的时候才会加锁，参考前面的文章，innodb要先扫描表中数据，被扫描到的行要加锁。

或者我们执行 `select` 的时候手动加上 排他锁 或者 共享锁，也会锁住。

这里老师讲到如果索引唯一键冲突，innodb为了做处理加了 `next_key lock (S)` 这个可以理解。

`insert .. select` 也是因为有 `select` 索引会加锁 也可以理解

问题：

图7那个死锁的案例，`session A` 的时候 只是执行了 `insert` 语句，执行 `insert` 的时候也没有`select`之类的，为什么也会在索引c上加个锁，是什么时候加的呢？？？是 `insert` 语句有索引的话都会给索引加锁么？？

2019-02-23

| 作者回复

不是都会，是在要写入的时候，发现有主键冲突，才会加上这个next-key lock的锁

2019-02-23



滔滔

0

老师，有个问题`insert into ... on duplicate key update`语句在发生冲突的时候是先加next key读锁，然后在执行后面的`update`语句时再给冲突记录加上写锁，从而把之前加的next key读锁变成了写锁，是这样的吗？

2019-02-21

作者回复

不是，发现冲突直接加的就是写锁

2019-02-24



王伯轩

0

内存锁 大大计划讲下么,实际中碰到内存锁被持有后一直不释放导致db直接crash掉

2019-02-18

作者回复

这个系列里没讲到了

这种我碰到比较多的是io压力特别大，导致有的事务执行不下去，但是占着锁

然后其他事务就拿不到锁，有一个600计时，超过就crash了

2019-02-18



信信

0

老师好，文中提到： `insert into t2(c,d) (select c+1, d from t force index(c) order by c desc limit 1`)的加锁范围是表 t 索引 c 上的 `(4,supremum]` 这个 next-key lock 和主键索引上 `id=4` 这一行。

可是如果我把表t的id为3这行先删除，再执行这个`insert...select`，那么别的会话执行`insert into t values(3,3,3)`会被阻塞，这说明4之前也是有间隙锁的？

另外，`select c+1, d from t force index(c) order by c desc limit 1 for update` 是不是不能用作等值查询那样分析？因为如果算等值查询，根据优化1是没有间隙锁的。

2019-02-17

作者回复

你说的对，这里其实是“向左扫描”，加锁范围应该是`(3,4]` 和 `(4, supremum]`。

0

2019-02-17



Justin

0

插入意向锁的gal lock和next key lock中的 gaplock互斥吗？

2019-02-15

作者回复

额，

这里我们要澄清一下哈

只有一个gap lock，就是 `next key lock = gap lock + record lock;`

我们说一个`insert`语句如果要插入一个间隙，而这个间隙上有`gap lock`的话，`insert`语句会被堵住，这个被堵住的效果，实现机制上是用插入意向锁和`gap lock`相互作用来实现的。

`gap lock`并不属于插入意向锁的一部分，就没有“插入意向锁的gal lock”这个概念哈

2019-02-16

41 | 怎么最快地复制一张表？

2019-02-15 林晓斌



我在上一篇文章最后，给你留下的问题是怎麽在两张表中拷贝数据。如果可以控制对源表的扫描行数和加锁范围很小的话，我们简单地使用`insert ...select`语句即可实现。

当然，为了避免对源表加读锁，更稳妥的方案是先将数据写到外部文本文件，然后再写回目标表。这时，有两种常用的方法。接下来的内容，我会和你详细展开一下这两种方法。

为了便于说明，我还是先创建一个表**db1.t**，并插入1000行数据，同时创建一个相同结构的表**db2.t**。

```
create database db1;
use db1;

create table t(id int primary key, a int, b int, index(a))engine=innodb;
delimiter ;;
create procedure idata()
begin
declare i int;
set i=1;
while(i<=1000)do
insert into t values(i,i,i);
set i=i+1;
end while;
end;;
delimiter ;
call idata();

create database db2;
create table db2.t like db1.t
```

假设，我们要把db1.t里面a>900的数据行导出来，插入到db2.t中。

mysqldump方法

一种方法是，使用mysqldump命令将数据导出成一组INSERT语句。你可以使用下面的命令：

```
mysqldump -h$host -P$port -u$user --add-locks=0 --no-create-info --single-transaction --set-gtid-purged=OFF db1
```

把结果输出到临时文件。

这条命令中，主要参数含义如下：

1. **-single-transaction**的作用是，在导出数据的时候不需要对表db1.t加表锁，而是使用**START TRANSACTION WITH CONSISTENT SNAPSHOT**的方法；
2. **-add-locks**设置为0，表示在输出的文件结果里，不增加"**LOCK TABLES t WRITE;**"；
3. **-no-create-info**的意思是，不需要导出表结构；

4. `-set-gtid-purged=off`表示的是，不输出跟GTID相关的信息；
5. `-result-file`指定了输出文件的路径，其中`client`表示生成的文件是在客户端机器上的。

通过这条`mysqldump`命令生成的`t.sql`文件中就包含了如图1所示的`INSERT`语句。

```
INSERT INTO `t` VALUES (901,901,901),(902,902,902),(903,903,903),(904,904,904),(905,905,905),(906,906,906),(907,907,907),(908,908,908),(909,909),(910,910,910),(911,911,911),(912,912,912),(913,913,913),(914,914,914),(915,915,915),(916,916,916),(917,917,917),(918,918,918),(919,919,919),(920,920,920),(921,921,921),(922,922,922),(923,923,923),(924,924,924),(925,925,925),(926,926,926),(927,927,927),(928,928,928),(929,929,929),(930,930,930),(931,931,931),(932,932,932),(933,933,933),(934,934,934),(935,935,935),(936,936,936),(937,937,937),(938,938,938)
```

图1 mysqldump输出文件的部分结果

可以看到，一条`INSERT`语句里面会包含多个`value`对，这是为了后续用这个文件来写入数据的时候，执行速度可以更快。

如果你希望生成的文件中一条`INSERT`语句只插入一行数据的话，可以在执行`mysqldump`命令时，加上参数`-skip-extended-insert`。

然后，你可以通过下面这条命令，将这些`INSERT`语句放到`db2`库里去执行。

```
mysql -h127.0.0.1 -P13000 -uroot db2 -e "source /client_tmp/t.sql"
```

需要说明的是，`source`并不是一条`SQL`语句，而是一个客户端命令。`mysql`客户端执行这个命令的流程是这样的：

1. 打开文件，默认以分号为结尾读取一条条的`SQL`语句；
2. 将`SQL`语句发送到服务端执行。

也就是说，服务端执行的并不是这个“`source t.sql`”语句，而是`INSERT`语句。所以，不论是在慢查询日志（`slow log`），还是在`binlog`，记录的都是这些要被真正执行的`INSERT`语句。

导出CSV文件

另一种方法是直接将结果导出成`.csv`文件。`MySQL`提供了下面的语法，用来将查询结果导出到服务端本地目录：

```
select * from db1.t where a>900 into outfile '/server_tmp/t.csv';
```

我们在使用这条语句时，需要注意如下几点。

1. 这条语句会将结果保存在服务端。如果你执行命令的客户端和`MySQL`服务端不在同一个机器上，客户端机器的临时目录下是不会生成`t.csv`文件的。

2. `into outfile`指定了文件的生成位置 (`/server_tmp/`)，这个位置必须受参数`secure_file_priv`的限制。参数`secure_file_priv`的可选值和作用分别是：

- 如果设置为`empty`，表示不限制文件生成的位置，这是不安全的设置；
- 如果设置为一个表示路径的字符串，就要求生成的文件只能放在这个指定的目录，或者它的子目录；
- 如果设置为`NULL`，就表示禁止在这个MySQL实例上执行`select ...into outfile`操作。

3. 这条命令不会帮你覆盖文件，因此你需要确保`/server_tmp/t.csv`这个文件不存在，否则执行语句时就会因为有同名文件的存在而报错。

4. 这条命令生成的文本文件中，原则上一个数据行对应文本文件的一行。但是，如果字段中包含换行符，在生成的文本中也会有换行符。不过类似换行符、制表符这类符号，前面都会跟上“\”这个转义符，这样就可以跟字段之间、数据行之间的分隔符区分开。

得到.csv导出文件后，你就可以用下面的`load data`命令将数据导入到目标表`db2.t`中。

```
load data infile '/server_tmp/t.csv' into table db2.t;
```

这条语句的执行流程如下所示。

1. 打开文件`/server_tmp/t.csv`，以制表符(`t`)作为字段间的分隔符，以换行符 (`\n`) 作为记录之间的分隔符，进行数据读取；
2. 启动事务。
3. 判断每一行的字段数与表`db2.t`是否相同：
 - 若不相同，则直接报错，事务回滚；
 - 若相同，则构造成一行，调用InnoDB引擎接口，写入到表中。
4. 重复步骤3，直到`/server_tmp/t.csv`整个文件读入完成，提交事务。

你可能有一个疑问，如果`binlog_format=statement`，这个`load`语句记录到`binlog`里以后，怎么在备库重放呢？

由于`/server_tmp/t.csv`文件只保存在主库所在的主机上，如果只是把这条语句原文写到`binlog`中，在备库执行的时候，备库的本地机器上没有这个文件，就会导致主备同步停止。

所以，这条语句执行的完整流程，其实是下面这样的。

1. 主库执行完成后，将`/server_tmp/t.csv`文件的内容直接写到`binlog`文件中。

2. 往binlog文件中写入语句`load data local infile '/tmp/SQL_LOAD_MB-1-0' INTO TABLE `db2`.`t`。`
3. 把这个binlog日志传到备库。
4. 备库的apply线程在执行这个事务日志时:
 - a. 先将binlog中t.csv文件的内容读出来，写入到本地临时目录`/tmp/SQL_LOAD_MB-1-0`中；
 - b. 再执行`load data`语句，往备库的`db2.t`表中插入跟主库相同的数据。

执行流程如图2所示：

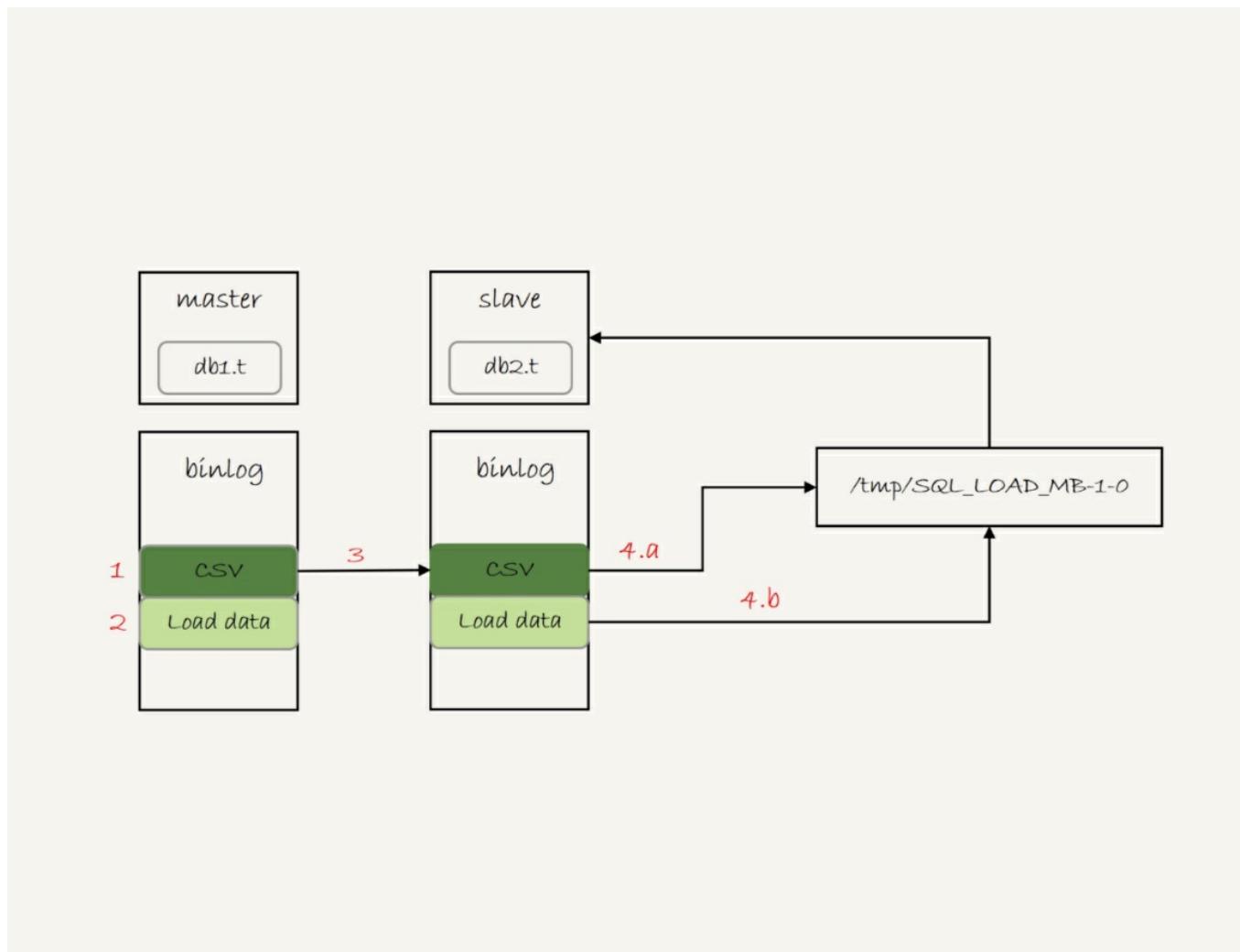


图2 load data的同步流程

注意，这里备库执行的`load data`语句里面，多了一个“`local`”。它的意思是“将执行这条命令的客户端所在机器的本地文件`/tmp/SQL_LOAD_MB-1-0`的内容，加载到目标表`db2.t`中”。

也就是说，`load data`命令有两种用法：

1. 不加“`local`”，是读取服务端的文件，这个文件必须在`secure_file_priv`指定的目录或子目录下；

- 加上“local”，读取的是客户端的文件，只要mysql客户端有访问这个文件的权限即可。这时候，MySQL客户端会先把本地文件传给服务端，然后执行上述的load data流程。

另外需要注意的是，**select ..into outfile**方法不会生成表结构文件，所以我们导数据时还需要单独的命令得到表结构定义。**mysqldump**提供了一个**-tab**参数，可以同时导出表结构定义文件和**CSV**数据文件。这条命令的使用方法如下：



```
mysqldump -h$host -P$port -u$user --single-transaction --set-gtid-purged=OFF db1 t --where="a>900" --tab=$secure_file_priv > t.sql & t.txt
```

这条命令会在**\$secure_file_priv**定义的目录下，创建一个**t.sql**文件保存建表语句，同时创建一个**t.txt**文件保存**CSV**数据。

物理拷贝方法

前面我们提到的**mysqldump**方法和导出**CSV**文件的方法，都是逻辑导数据的方法，也就是将数据从表**db1.t**中读出来，生成文本，然后再写入目标表**db2.t**中。

你可能会问，有物理导数据的方法吗？比如，直接把**db1.t**表的**.frm**文件和**.ibd**文件拷贝到**db2**目录下，是否可行呢？

答案是不行的。

因为，一个**InnoDB**表，除了包含这两个物理文件外，还需要在数据字典中注册。直接拷贝这两个文件的话，因为数据字典中没有**db2.t**这个表，系统是不会识别和接受它们的。

不过，在**MySQL 5.6**版本引入了可传输表空间(**transportable tablespace**)的方法，可以通过导出+导入表空间的方式，实现物理拷贝表的功能。

假设我们现在的目标是在**db1**库下，复制一个跟表**t**相同的表**r**，具体的执行步骤如下：

1. 执行 **create table r like t**，创建一个相同表结构的空表；
2. 执行 **alter table r discard tablespace**，这时候**r.ibd**文件会被删除；
3. 执行 **flush table t for export**，这时候**db1**目录下会生成一个**t.cfg**文件；
4. 在**db1**目录下执行 **cp t.cfg r.cfg; cp t.ibd r.ibd;** 这两个命令（这里需要注意的是，拷贝得到的两个文件，MySQL进程要有读写权限）；
5. 执行 **unlock tables**，这时候**t.cfg**文件会被删除；
6. 执行 **alter table r import tablespace**，将这个**r.ibd**文件作为表**r**的新的表空间，由于这个文件

的数据内容和t.ibd是相同的，所以表r中就有了和表t相同的数据。

至此，拷贝表数据的操作就完成了。这个流程的执行过程图如下：

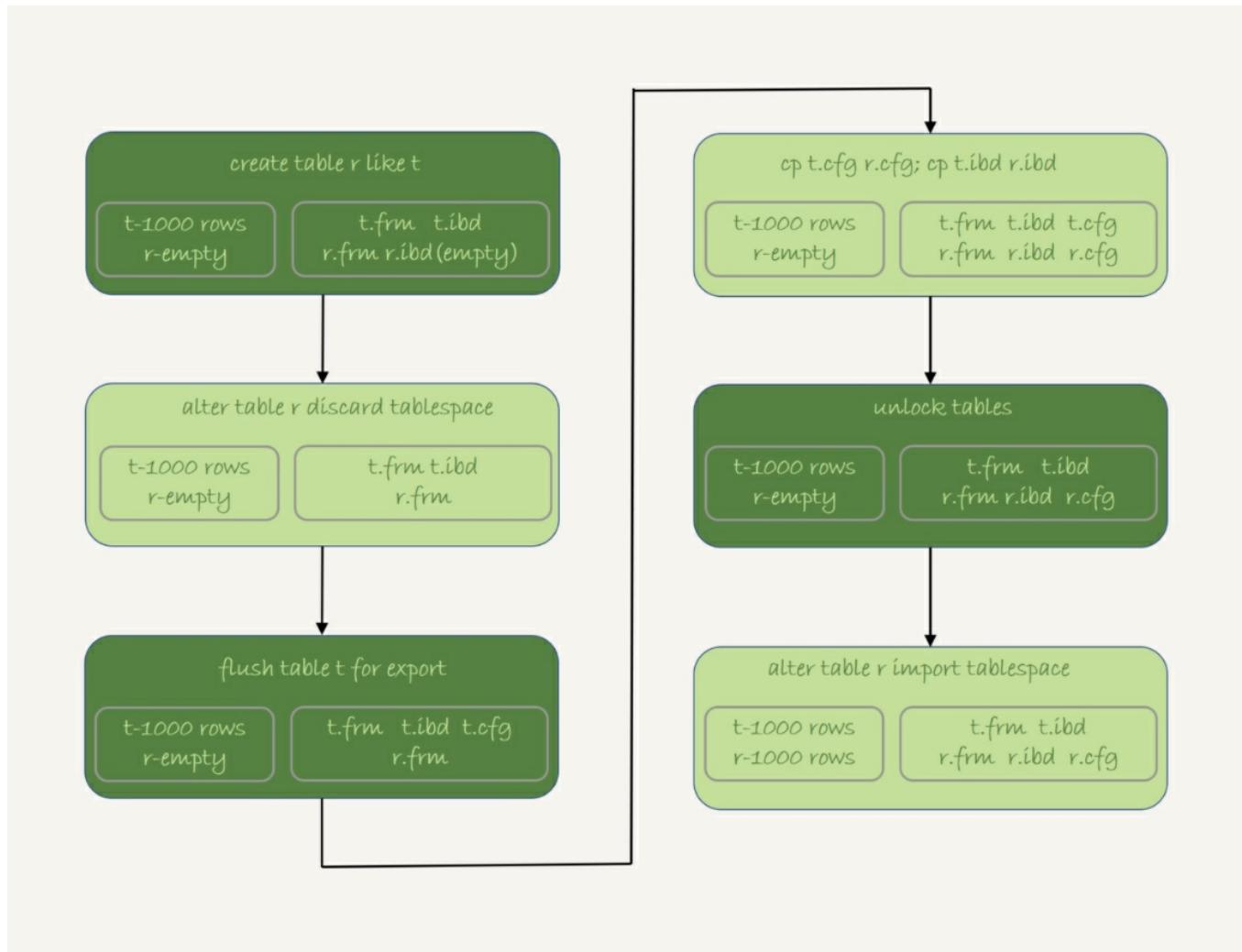


图3 物理拷贝表

关于拷贝表的这个流程，有以下几个注意点：

1. 在第3步执行完`flsuh table`命令之后，`db1.t`整个表处于只读状态，直到执行`unlock tables`命令后才释放读锁；
2. 在执行`import tablespace`的时候，为了让文件里的表空间id和数据字典中的一致，会修改`r.ibd`的表空间id。而这个表空间id存在于每一个数据页中。因此，如果是一个很大的文件（比如TB级别），每个数据页都需要修改，所以你会看到这个`import`语句的执行是需要一些时间的。当然，如果是相比于逻辑导入的方法，`import`语句的耗时是非常短的。

小结

今天这篇文章，我和你介绍了三种将一个表的数据导入到另外一个表中的方法。

我们来对比一下这三种方法的优缺点。

- 物理拷贝的方式速度最快，尤其对于大表拷贝来说是最快的方法。如果出现误删表的情况，用备份恢复出误删之前的临时库，然后再把临时库中的表拷贝到生产库上，是恢复数据最快的方法。但是，这种方法的使用也有一定的局限性：
 - 必须是全表拷贝，不能只拷贝部分数据；
 - 需要到服务器上拷贝数据，在用户无法登录数据库主机的场景下无法使用；
 - 由于是通过拷贝物理文件实现的，源表和目标表都是使用InnoDB引擎时才能使用。
- 用mysqldump生成包含INSERT语句文件的方法，可以在where参数增加过滤条件，来实现只导出部分数据。这个方式的不足之一是，不能使用join这种比较复杂的where条件写法。
- 用select ...into outfile的方法是最灵活的，支持所有的SQL写法。但，这个方法的缺点之一就是，每次只能导出一张表的数据，而且表结构也需要另外的语句单独备份。

后两种方式都是逻辑备份方式，是可以跨引擎使用的。

最后，我给你留下一个思考题吧。

我们前面介绍binlog_format=statement的时候，binlog记录的load data命令是带local的。既然这条命令是发送到备库去执行的，那么备库执行的时候也是本地执行，为什么需要这个local呢？如果写到binlog中的命令不带local，又会出现什么问题呢？

你可以把你的分析写在评论区，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上篇文章最后给你留下的思考题，已经在今天这篇文章的正文部分做了回答。

上篇文章的评论区有几个非常好的留言，我在这里和你分享一下。

@huolang 同学提了一个问题：如果sessionA拿到c=5的记录锁是写锁，那为什么sessionB和sessionC还能加c=5的读锁呢？

这是因为next-key lock是先加间隙锁，再加记录锁的。加间隙锁成功了，加记录锁就会被堵住。如果你对这个过程有疑问的话，可以再复习一下[第30篇文章](#)中的相关内容。

@一大只 同学做了一个实验，验证了主键冲突以后，insert语句加间隙锁的效果。比我在上篇文章正文中提的那个回滚导致死锁的例子更直观，体现了他对这个知识点非常好的理解和思考，很赞。

@roaming 同学验证了在MySQL 8.0版本中，已经能够用临时表处理insert ...select写入原表的语句了。

@老杨同志 的回答提到了我们本文中说到的几个方法。

The image shows a promotional banner for a MySQL course. At the top left is the '极客时间' logo. The main title 'MySQL 实战 45 讲' is displayed prominently in large, bold, dark font. Below it is a subtitle '从原理到实战，丁奇带你搞懂 MySQL' in a smaller, lighter font. To the right of the text is a portrait of the instructor, Lin Xiaobin (丁奇), a young man with glasses and a black shirt, with his arms crossed. On the left side of the banner, there is a section with the name '林晓斌' (Lin Xiaobin) and the text '网名丁奇 前阿里资深技术专家'. At the bottom of the banner, there is a call-to-action: '新版升级：点击「请朋友读」，10位好友免费读，邀请订阅更有现金奖励。' (New version upgrade: Click 'Invite friends to read', get 10 free reads, and there are cash rewards for subscriptions).

精选留言

-  **poppy** 4
关于思考题，我理解是备库的同步线程其实相当于备库的一个客户端，由于备库的会把binlog中t.csv的内容写到/tmp/SQL_LOAD_MB-1-0中，如果load data命令不加'local'表示读取服务端的文件，文件必须在secure_file_priv指定的目录或子目录，此时可能找不到该文件，主备同步执行会失败。而加上local的话，表示读取客户端的文件，既然备份线程都能在该目录下创建临时文件/tmp/SQL_LOAD_MB-1-0,必然也有权限访问，把该文件传给服务端执行。
- 2019-02-15
- | 作者回复
-  **丶这是其中一个原因**
2019-02-16
-  **☆apple** 3
通知对方更新数据的意思是：针对事务内的3个操作：插入和更新两个都是本地操作，第三个操作是远程调用，这里远程调用其实是想把本地操作的那两条通知对方（对方：远程调用），让对方把数据更新，这样双方（我和远程调用方）的数据达到一致，如果对方操作失败，事务的前两个操作也会回滚，主要是想保证双方数据的一致性，因为远程调用可能会出现网络延迟超时等因素，极端情况会导致事务10s左右才能处理完毕，想问的是这样耗时的事务会带来哪些影响呢？
- 设计的初衷是想这三个操作能原子执行，只要有不成功就可以回滚，保证两方数据的一致性

耗时长的远程调用不放在事务中执行，会出现我这面数据完成了，而对方那面由于网络等问题，并没有更新，这样两方的数据就出现不一致了

2019-02-15

| 作者回复

嗯 了解了

这种设计我觉得就是会对并发性有比较大的影响。

一般如果网络状态不好的，会建议把这个更新操作放到消息队列。

就是说

1. 先本地提交事务。

2. 把通知这个动作放到消息队列，失败了可以重试；

3. 远端接收事件要设置成可重入的，就是即使同一个消息收到两次，也跟收到一次是相同的效果。

2 和3 配合起来保证最终一致性。

这种设计我见到得比较多，你评估下是否符合你们业务的需求哈

2019-02-15



undefined

3

老师，用物理导入的方式执行 `alter table r import tablespace` 时 提示ERROR 1812 (HY000): Tablespace is missing for table `db1`.`r`。此时 db1/ 下面的文件有 db.opt r.cfg r.frm r.ibd t.frm t.ibd；这个该怎么处理

执行步骤：

```
mysql> create table r like t;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> alter table r discard tablespace;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> flush table t for export;
Query OK, 0 rows affected (0.00 sec)
```

```
cp t.cfg r.cfg
cp t.ibd r.ibd
```

```
mysql> unlock tables;
```

Query OK, 0 rows affected (0.01 sec)

```
mysql> alter table r import tablespace;
ERROR 1812 (HY000): Tablespace is missing for table `db1`.`r`.
```

2019-02-15

作者回复

应该就是评论区其他同学帮忙回复的权限问题了吧?

2019-02-15



lionetes

2

```
mysql> select * from t;
```

id	name
1	Bob
2	Mary
3	Jane
4	Lisa
5	Mary
6	Jane
7	Lisa

7 rows in set (0.00 sec)

```
mysql> create table tt like t;
```

Query OK, 0 rows affected (0.03 sec)

```
mysql> alter table tt discard tablespace;
```

Query OK, 0 rows affected (0.01 sec)

```
mysql> flush table t for export;
```

Query OK, 0 rows affected (0.01 sec)

```
mysql> unlock tables;
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> alter table tt import tablespace;
```

Query OK, 0 rows affected (0.03 sec)

```
mysql> show tables;
```

+-----+

| Tables_in_test |

```
+-----+
| t |
| t2 |
| tt |
+-----+
3 rows in set (0.00 sec)
```

```
mysql> select * from t;
+----+----+
| id | name |
+----+----+
| 1 | Bob |
| 2 | Mary |
| 3 | Jane |
| 4 | Lisa |
| 5 | Mary |
| 6 | Jane |
| 7 | Lisa |
+----+----+
7 rows in set (0.00 sec)
```

```
mysql> select * from tt;
+----+----+
| id | name |
+----+----+
| 1 | Bob |
| 2 | Mary |
| 3 | Jane |
| 4 | Lisa |
| 5 | Mary |
| 6 | Jane |
| 7 | Lisa |
+----+----+
7 rows in set (0.00 sec)
```

|| 后 查看 tt.cfg 文件没有自动删除 5.7mysql

```
-rw-r----. 1 mysql mysql 380 2月 15 09:51 tt.cfg
-rw-r----. 1 mysql mysql 8586 2月 15 09:49 tt.frm
-rw-r----. 1 mysql mysql 98304 2月 15 09:51 tt.ibd
```

| 作者回复

你说得对，细致

import动作 不会自动删除cfg文件，我图改一下

2019-02-15



☆apple う

2

老师，我想问一个关于事务的问题，一个事务中有3个操作，插入一条数据(本地操作),更新一条数据(本地操作)，然后远程调用，通知对方更新上面数据(如果远程调用失败会重试，最多3次，如果遇到网络等问题，远程调用时间会达到5s,极端情况3次会达到15s)，那么极端情况事务将长达5-15s，这样会带来什么影响吗？

2019-02-15

| 作者回复

“通知对方更新上面数据”是啥概念，如果你这个事务没提交，其他线程也看不到前两个操作的结果的。

设计上不建议留这么长的事务哈，最好是可以先把事务提交了，再去做耗时的操作。

2019-02-15



AstonPutting

1

老师，mysqlpump能否在平时代替mysqldump的使用？

2019-02-22

| 作者回复

我觉得是

2019-02-23



PengfeiWang

1

老师，您好：

文中“**-add-locks** 设置为 0，表示在输出的文件结果里，不增加"LOCK TABLES t WRITE;" 是否是笔误，**--add-locks**应该是在**insert**语句前后添加锁，我的理解此处应该是**--skip-add-locks**，不知道是否是这样？

2019-02-18

| 作者回复

嗯嗯，命令中写错了，是**--add-locks=0**,

效果上跟**--skip-add-locks**是一样的哈

细致

2019-02-19



长杰

1

课后题答案

不加“local”，是读取服务端的文件，这个文件必须在 **secure_file_priv** 指定的目录或子目录下；而备库的**apply**线程执行时先讲**csv**内容读出生成**tmp**目录下的临时文件，这个目录容易受**secure_file_priv**的影响，如果备库改参数设置为Null或指定的目录，可能导致**load**操作失败，加**local**则

不受这个影响。

2019-02-17

| 作者回复

0

2019-02-18



尘封

1

老师mysqldump导出的文件里，单条sql里的value值有什么限制吗默认情况下，假如一个表有几百万，那mysql会分为多少个sql导出？

问题：因为从库可能没有load的权限，所以local

2019-02-15

| 作者回复

好问题，

会控制单行不会超过参数net_buffer_length，这个参数是可以通过--net_buffer_length 传给mysql dump 工具的

2019-02-28



佳

0

老师好，这个/tmp/SQL_LOAD_MB-1-0 是应该在主库上面，还是备库上面？为啥我执行完是在主库上面出现了这个文件呢？

2019-03-14

| 作者回复

就是在MySQL的运行进程所在的主机上

2019-03-16



xxj123go

0

传输表空间方式对主从同步会有影响么

2019-03-12

| 作者回复

你可以看下执行以后，进不进binlog 0

2019-03-13



王显伟

0

第一位留言的朋友报错我也复现了，原因是用root复制的文件，没有修改属组导致的

2019-02-16

| 作者回复

0

2019-02-17



夜空中最亮的星（华仔）

0

学习完老师的课都想做dba了

2019-02-15



undefined

0

老师 错误信息的截屏 <https://www.dropbox.com/s/8wyet4bt9yfjsau/mysqlerror.png?dl=0>

MySQL 5.7, Mac 上的 Docker 容器里面跑的, 版本是 5.7.17

2019-02-15

| 作者回复

额, 打不开。。

可否发个微博贴图

2019-02-16



晨思暮语

0

不好意思, 第一条留言中, 实验三的最后一天语句还是少了, 在这里贴一下,

mysql> select * from t where id=1;

+----+-----+

| id | a |

+----+-----+

| 1 | 3 |

+----+-----+

1 row in set (0.00 sec)

2019-02-15



晨思暮语

0

老师好, 由于字数限制, 分两条:

我用的是percona数据库, 问题是第15章中的思考题。

根据我做的实验, 结论应该是:

MySQL 调用了 InnoDB 引擎提供的“修改为 (1,2)”这个接口, 但是引擎发现值与原来相同, 不更新, 直接返回

一直没有想明白, 老师再帮忙看看, 谢谢!

2019-02-15

| 作者回复

我两个留言连在一起看没看明白你对哪个步骤的那个结果有疑虑,

可以写在现象里面 (用注释即可) 哈

2019-02-16



晨思暮语

0

mysql> select version();

+-----+

| version() |

+-----+

| 5.7.22-log |

+-----+

实验1:

SESSION A:

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from t where id=1;
```

```
+----+-----+
```

```
| id | a |
```

```
+----+-----+
```

```
| 1 | 2 |
```

```
+----+-----+
```

```
1 row in set (0.00 sec)
```

SESSION B:

```
mysql> update t set a=3 where id=1;
```

```
Query OK, 1 row affected (0.01 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

SESSION A:

```
mysql> update t set a=3 where id=1;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
Rows matched: 1 Changed: 0 Warnings: 0
```

```
mysql> select * from t where id=1;
```

```
+----+-----+
```

```
| id | a |
```

```
+----+-----+
```

```
| 1 | 2 |
```

```
+----+-----+
```

```
1 row in set (0.00 sec)
```

实验2:

SESSION A:

```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from t where id=1;
```

```
+----+-----+
```

```
| id | a |
```

```
+----+-----+
```

```
| 1 | 2 |
```

```
+----+-----+
```

1 row in set (0.00 sec)

SESSION B:

mysql> update t set a=3 where id=1;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

SESSION A:

mysql> update t set a=3 where id=1;

BLOCKED

SESSION B:

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

SESSION A:UPDATE

mysql> update t set a=3 where id=1;

Query OK, 0 rows affected (5.43 sec)

Rows matched: 1 Changed: 0 Warnings: 0

mysql>

mysql> select * from t where id=1;

+----+-----+

| id | a |

+----+-----+

| 1 | 2 |

+----+-----+

1 row in set (0.00 sec)

实验3:

SESSION A:

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select * from t where id=1;

+----+-----+

| id | a |

+----+-----+

| 1 | 2 |

+----+-----+

1 row in set (0.00 sec)

SESSION B:

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

```
mysql> update t set a=3 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
SESSION A:
mysql> update t set a=3 where id=1;
blocked
SESSION B:
mysql> rollback;
Query OK, 0 rows affected (0.00 sec)
```

SESSION A:UPDATE

```
mysql> update t set a=3 where id=1;
Query OK, 1 row affected (5.21 sec)
Rows matched: 1 C
```

2019-02-15



库淘淘

0

如果不加local 如secure_file_priv 设置为null 或者路径 可能就不能成功,这样加了之后可以保证执行成功率不受参数secure_file_priv影响。还有发现物理拷贝文件后，权限所属用户还得改下，不然import tablespace 会报错找不到文件，老师是不是应该补充上去，不然容易踩坑。

2019-02-15

| 作者回复

嗯嗯，有同学已经踩了，
我加个说明进去，多谢提醒

2019-02-15



lionetes

0

@undefined 看下是否是 权限问题引起的 cp 完后 是不是mysql 权限

2019-02-15

| 作者回复

经验丰富

如果进程用mysql用户启动，命令行是在root账号下，确实会出现这种情况

2019-02-15



Ryoma

0

问老师一个主题无关的问题：现有数据库中有个表字段为text类型，但是目前发现text中的数据有点不太对。

请问在MySQL中有没有办法确认在插入时是否发生截断数据的情况么？（因为该字段被修改过，我现在不方便恢复当时的现场）

2019-02-15

| 作者回复

看那个语句的binlog （是row吧？） 』

2019-02-15

42 | grant之后要跟着flush privileges吗？

2019-02-18 林晓斌



在MySQL里面，grant语句是用来给用户赋权的。不知道你有没有见过一些操作文档里面提到，grant之后要马上跟着执行一个flush privileges命令，才能使赋权语句生效。我最开始使用MySQL的时候，就是照着一个操作文档的说明按照这个顺序操作的。

那么，grant之后真的需要执行flush privileges吗？如果没有执行这个flush命令的话，赋权语句真的不能生效吗？

接下来，我就先和你介绍一下grant语句和flush privileges语句分别做了什么事情，然后再一起来分析这个问题。

为了便于说明，我先创建一个用户：

```
create user 'ua'@'%' identified by 'pa';
```

这条语句的逻辑是创建一个用户'ua'@'%', 密码是pa。注意，在MySQL里面，用户名(user)+地址(host)才表示一个用户，因此 ua@ip1 和 ua@ip2 代表的是两个不同的用户。

这条命令做了两个动作：

1. 磁盘上，往mysql.user表里插入一行，由于没有指定权限，所以这行数据上所有表示权限的字段的值都是N；

2. 内存里，往数组acl_users里插入一个acl_user对象，这个对象的access字段值为0。

图1就是这个时刻用户ua在user表中的状态。

```
mysql> select * from mysql.user where user='ua'\G
***** 1. row *****
      Host: %
      User: ua
  Select_priv: N
 Insert_priv: N
Update_priv: N
Delete_priv: N
Create_priv: N
   Drop_priv: N
 Reload_priv: N
Shutdown_priv: N
Process_priv: N
   File_priv: N
  Grant_priv: N
References_priv: N
   Index_priv: N
    Alter_priv: N
 Show_db_priv: N
   Super_priv: N
Create_tmp_table_priv: N
 Lock_tables_priv: N
 Execute_priv: N
Repl_slave_priv: N
Repl_client_priv: N
Create_view_priv: N
 Show_view_priv: N
Create_routine_priv: N
 Alter_routine_priv: N
Create_user_priv: N
   Event_priv: N
 Trigger_priv: N
Create_tablespace_priv: N
```

图1 mysql.user 数据行

在MySQL中，用户权限是有不同的范围的。接下来，我就按照用户权限范围从大到小的顺序依次和你说明。

全局权限

全局权限，作用于整个MySQL实例，这些权限信息保存在mysql库的user表里。如果我要给用户

ua赋一个最高权限的话，语句是这么写的：

```
grant all privileges on *.* to 'ua'@'%' with grant option;
```

这个**grant**命令做了两个动作：

1. 磁盘上，将**mysql.user**表里，用户'ua'@'%'这一行的所有表示权限的字段的值都修改为‘Y’；
2. 内存里，从数组**acl_users**中找到这个用户对应的对象，将**access**值（权限位）修改为二进制的“全1”。

在这个**grant**命令执行完成后，如果有新的客户端使用用户名**ua**登录成功，**MySQL**会为新连接维护一个线程对象，然后从**acl_users**数组里查到这个用户的权限，并将权限值拷贝到这个线程对象中。之后在这个连接中执行的语句，所有关于全局权限的判断，都直接使用线程对象内部保存的权限位。

基于上面的分析我们可以知道：

1. **grant** 命令对于全局权限，同时更新了磁盘和内存。命令完成后即时生效，接下来新创建的连接会使用新的权限。
2. 对于一个已经存在的连接，它的全局权限不受**grant**命令的影响。

需要说明的是，一般在生产环境上要合理控制用户权限的范围。我们上面用到的这个**grant**语句就是一个典型的错误示范。如果一个用户有所有权限，一般就不应该设置为所有IP地址都可以访问。

如果要收回上面的**grant**语句赋予的权限，你可以使用下面这条命令：

```
revoke all privileges on *.* from 'ua'@'%';
```

这条**revoke**命令的用法与**grant**类似，做了如下两个动作：

1. 磁盘上，将**mysql.user**表里，用户'ua'@'%'这一行的所有表示权限的字段的值都修改为“N”；
2. 内存里，从数组**acl_users**中找到这个用户对应的对象，将**access**的值修改为0。

db权限

除了全局权限，**MySQL**也支持库级别的权限定义。如果要让用户**ua**拥有库**db1**的所有权限，可以执行下面这条命令：

```
grant all privileges on db1.* to 'ua'@'%' with grant option;
```

基于库的权限记录保存在mysql.db表中，在内存里则保存在数组acl_dbs中。这条grant命令做了如下两个动作：

1. 磁盘上，往mysql.db表中插入了一行记录，所有权限位字段设置为“Y”；
2. 内存里，增加一个对象到数组acl_dbs中，这个对象的权限位为“全1”。

图2就是这个时刻用户ua在db表中的状态。

```
mysql> select * from mysql.db where user='ua'\G
***** 1. row *****
      Host: %
      Db: db1
    User: ua
Select_priv: Y
Insert_priv: Y
Update_priv: Y
Delete_priv: Y
Create_priv: Y
Drop_priv: Y
Grant_priv: Y
References_priv: Y
Index_priv: Y
Alter_priv: Y
Create_tmp_table_priv: Y
Lock_tables_priv: Y
Create_view_priv: Y
Show_view_priv: Y
Create_routine_priv: Y
Alter_routine_priv: Y
Execute_priv: Y
Event_priv: Y
Trigger_priv: Y
1 row in set (0.00 sec)
```

图2 mysql.db 数据行

每次需要判断一个用户对一个数据库读写权限的时候，都需要遍历一次acl_dbs数组，根据user、host和db找到匹配的对象，然后根据对象的权限位来判断。

也就是说，grant修改db权限的时候，是同时对磁盘和内存生效的。

`grant`操作对于已经存在的连接的影响，在全局权限和基于db的权限效果是不同的。接下来，我们做一个对照试验来分别看一下。

	session A	session B	session C
T1	<pre>connect(root,root) create database db1; create user 'ua'@'%' identified by 'pa'; grant super on ** to 'ua'@'%'; grant all privileges on db1.* to 'ua'@'%';</pre>		
T2		<pre>connect(ua,pa) set global sync_binlog=1; (Query OK) create table db1.t(c int); (Query OK)</pre>	<pre>connect(ua,pa) use db1;</pre>
T3	<pre>revoke super on ** from 'ua'@'%';</pre>		
T4		<pre>set global sync_binlog=1; (Query OK) alter table db1.t engine=innodb; (Query OK)</pre>	<pre>alter table t engine=innodb; (Query OK)</pre>
T5	<pre>revoke all privileges on db1.* from 'ua'@'%';</pre>		
T6		<pre>set global sync_binlog=1; (Query OK) alter table db1.t engine=innodb; (ALTER command denied)</pre>	<pre>alter table t engine=innodb; (Query OK)</pre>

图3 权限操作效果

需要说明的是，图中`set global sync_binlog`这个操作是需要`super`权限的。

可以看到，虽然用户ua的`super`权限在T3时刻已经通过`revoke`语句回收了，但是在T4时刻执行`set global`的时候，权限验证还是通过了。这是因为`super`是全局权限，这个权限信息在线程对象中，而`revoke`操作影响不到这个线程对象。

而在T5时刻去掉ua对db1库的所有权限后，在T6时刻session B再操作db1库的表，就会报错“权限不足”。这是因为`acl_dbs`是一个全局数组，所有线程判断db权限都用这个数组，这样`revoke`操

作马上就会影响到**session B**。

这里在代码实现上有一个特别的逻辑，如果当前会话已经处于某一个**db**里面，之前**use**这个库的时候拿到的库权限会保存在会话变量中。

你可以看到在**T6**时刻，**session C**和**session B**对表**t**的操作逻辑是一样的。但是**session B**报错，而**session C**可以执行成功。这是因为**session C**在**T2**时刻执行的**use db1**，拿到了这个库的权限，在切换出**db1**库之前，**session C**对这个库就一直有权限。

表权限和列权限

除了**db**级别的权限外，MySQL支持更细粒度的表权限和列权限。其中，表权限定义存放在表**mysql.tables_priv**中，列权限定义存放在表**mysql.columns_priv**中。这两类权限，组合起来存放在内存的**hash**结构**column_priv_hash**中。

这两类权限的赋权命令如下：

```
create table db1.t1(id int, a int);

grant all privileges on db1.t1 to 'ua'@'%' with grant option;
GRANT SELECT(id), INSERT (id,a) ON mydb.mytbl TO 'ua'@'%' with grant option;
```

跟**db**权限类似，这两个权限每次**grant**的时候都会修改数据表，也会同步修改内存中的**hash**结构。因此，对这两类权限的操作，也会马上影响到已经存在的连接。

看到这里，你一定会问，看来**grant**语句都是即时生效的，那这么看应该就不需要执行**flush privileges**语句了呀。

答案也确实是这样的。

flush privileges命令会清空**acl_users**数组，然后从**mysql.user**表中读取数据重新加载，重新构造一个**acl_users**数组。也就是说，以数据表中的数据为准，会将全局权限内存数组重新加载一遍。

同样地，对于**db**权限、表权限和列权限，MySQL也做了这样的处理。

也就是说，如果内存的权限数据和磁盘数据表相同的话，不需要执行**flush privileges**。而如果我们都是用**grant/revoke**语句来执行的话，内存和数据表本来就是保持同步更新的。

因此，正常情况下，**grant**命令之后，没有必要跟着执行**flush privileges**命令。

flush privileges使用场景

那么，**flush privileges**是在什么时候使用呢？显然，当数据表中的权限数据跟内存中的权限数据不一致的时候，**flush privileges**语句可以用来重建内存数据，达到一致状态。

这种不一致往往是由不规范的操作导致的，比如直接用**DML**语句操作系统权限表。我们来看一下下面这个场景：

	client A	client B
T1	connect(root, root) create user 'ua'@'%' identified by 'pa';	
T2		connect(ua,pa) (connect ok) disconnect
T3	delete from mysql.user where user='ua';	
T4		connect(ua,pa) (connect ok) disconnect
T5	flush privileges;	
T6		connect(ua,pa) (Access Denied)

图4 使用**flush privileges**

可以看到，T3时刻虽然已经用**delete**语句删除了用户ua，但是在T4时刻，仍然可以用ua连接成功。原因就是，这时候内存中**acl_users**数组中还有这个用户，因此系统判断时认为用户还正常存在。

在T5时刻执行过**flush**命令后，内存更新，T6时刻再要用ua来登录的话，就会报错“无法访问”了。

直接操作系统表是不规范的操作，这个不一致状态也会导致一些更“诡异”的现象发生。比如，前面这个通过**delete**语句删除用户的例子，就会出现下面的情况：

	client A
T1	connect(root, root) create user 'ua'@'%' identified by 'pa';
T2	
T3	delete from mysql.user where user='ua';
T4	grant super on ** to 'ua'@'%' with grant option; ERROR 1133 (42000): Can't find any matching row in the user table
T5	create user 'ua'@'%' identified by 'pa'; ERROR 1396 (HY000): Operation CREATE USER failed for 'ua'@'%'

图5 不规范权限操作导致的异常

可以看到，由于在T3时刻直接删除了数据表的记录，而内存的数据还存在。这就导致了：

1. T4时刻给用户ua赋权限失败，因为mysql.user表中找不到这行记录；
2. 而T5时刻要重新创建这个用户也不行，因为在做内存判断的时候，会认为这个用户还存在。

小结

今天这篇文章，我和你介绍了MySQL用户权限在数据表和内存中的存在形式，以及grant和revoke命令的执行逻辑。

grant语句会同时修改数据表和内存，判断权限的时候使用的是内存数据。因此，规范地使用grant和revoke语句，是不需要随后加上flush privileges语句的。

flush privileges语句本身会用数据表的数据重建一份内存权限数据，所以在权限数据可能存在不一致的情况下再使用。而这种不一致往往是由于直接用DML语句操作系统权限表导致的，所以我们尽量不要使用这类语句。

另外，在使用grant语句赋权时，你可能还会看到这样的写法：

```
grant super on *.* to 'ua'@'%' identified by 'pa';
```

这条命令加了**identified by**‘密码’，语句的逻辑里面除了赋权外，还包含了：

1. 如果用户'ua'@'%'不存在，就创建这个用户，密码是pa;
2. 如果用户ua已经存在，就将密码修改成pa。

这也是一种不建议的写法，因为这种写法很容易就会不慎把密码给改了。

“grant之后随手加flush privileges”，我自己是这么使用了两三年之后，在看代码的时候才发现其实并不需要这样做，那已经是2011年的事情了。

去年我看到一位小伙伴这么操作的时候，指出这个问题时，他也觉得很神奇。因为，他和我一样看的第一份文档就是这么写的，自己也一直是这么用的。

所以，今天的课后问题是，请你也来说一说，在使用数据库或者写代码的过程中，有没有遇到过类似的场景：误用了很长时间以后，由于一个契机发现“啊，原来我错了这么久”？

你可以把你的经历写在留言区，我会在下一篇文章的末尾选取有趣的评论和你分享。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，MySQL解析statement格式的binlog的时候，对于load data命令，解析出来为什么用的是load data local。

这样做的一个原因是，为了确保备库应用binlog正常。因为备库可能配置了secure_file_priv=null，所以如果不使用local的话，可能会导入失败，造成主备同步延迟。

另一种应用场景是使用mysqlbinlog工具解析binlog文件，并应用到目标库的情况。你可以使用下面这条命令：

```
mysqlbinlog $binlog_file | mysql -h$host -P$port -u$user -p$pwd
```

把日志直接解析出来发给目标库执行。增加local，就能让这个方法支持非本地的\$host。

评论区留言点赞板：

@poppy、@库淘淘 两位同学提到了第一个场景；

@王显伟 @lionetes 两位同学帮忙回答了 @undefined 同学的疑问，拷贝出来的文件要确保 MySQL进程可以读。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「」，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



undefined

7

权限的作用范围和修改策略总结：

<http://ww1.sinaimg.cn/large/d1885ed1ly1g0ab2twmjaj21gs0js78u.jpg>

2019-02-18

| 作者回复

[], 优秀

2019-02-18



夜空中最亮的星 (华仔)

3

通过老师的讲解 `flush privileges` 这回彻底懂了，高兴[]

2019-02-18

| 作者回复

[]

2019-02-19



way

1

写个比较小的点：在命令行查询数据需要行转列的时候习惯加个\G；比如`slave slave stauts \G`；后来发现；是多余的。列几个常用的

\G 行转列并发送给 mysql server

\g 等同于；

\! 执行系统命令

\q exit
\c 清除当前SQL(不执行)
\s mysql status 信息
其他参考 \h

2019-02-20

| 作者回复

[]

我最开始使用MySQL的时候，就是不自然的在\G后面加分号
而且还看到报错，好紧张

2019-02-20

 XD

1

老师，我刚说的是acl_db，是在db切换的时候，从acl_dbs拷贝到线程内部的？类似acl_user。

session a
drop user 'test'@'%';
create user 'test'@'%' identified by '123456';
grant SELECT,UPDATE on gt.* to 'test'@'%';

session b 使用test登录

use gt;

session a
revoke SELECT,UPDATE on gt.* from 'test'@'%';

session b
show databases; //只能看到information_schema库
use gt; // Access denied for user 'test'@'%' to database 'gt'
show tables; //可以看到gt库中所有的表
select/update //操作都正常

2019-02-18

| 作者回复

你说的对，我刚翻代码确认了下，确实是特别对“当前db”有一个放过的逻辑。

多谢指正。我勘误下。

2019-02-19



夹心面包

1

我在此分享一个授权库的小技巧，如果需要授权多个库，库名还有规律，比如 db_201701 db_201702

可以采用正则匹配写一条 grant on db_____,每一个_代表一个字符。这样避免了多次授权，简化了过程。我们线上已经采用

2019-02-18

| 作者回复

是的，MySQL还支持 % 赋权，%表示匹配任意字符串，

比如

grant all privileges on `db%`.* to ... 表示所有以db为前缀的库。

不过。。。我比较不建议这么用

2019-02-19



萤火虫

0

坚持到最后 为老师打call

2019-02-20

| 作者回复

[]

是真爱

2019-02-20



wljs

0

老师我想问个问题 我们公司一个订单表有110个字段 想拆分成两个表 第一个表放经常查的字段 第二个表放不常查的 现在程序端不想改sql，数据库端来实现 当查询字段中 第一个表不存在 就去关联第二个表查出数据 db能实现不？

2019-02-19



舜

0

老师，介绍完了order by后能不能继续介绍下group by的原理？等了好久了，一直想继续在order by基础上理解下group by，在使用过程中两者在索引利用上很相近，性能考虑也类似

2019-02-19

| 作者回复

37篇讲了group by的，你看下

还有问再提出来

2019-02-19



旭东

0

老师请教一个问题：MySQL 表设计时列表顺序对MySQL性能的影响大吗？对表的列顺序有什么建议吗？

2019-02-18

| 作者回复

没有影响

建议就是每次如果要加列都加到最后一列

2019-02-19



XD

0

老师，实际测试了下。

两个会话ab，登陆账号都为user。a中给user授予db1的select、update权限，b切换到db1，可以正常增改。然后a中回收该用户的db权限，b会话中的用户还是可以进行增改操作的。

我发现用户的db权限好像是在切换数据库的时候刷新的，只要不切换，grant操作并不会产生作用，所以acl_db是否也是维护在线程内部的呢？

以及，权限检验应该是在优化器的语义分析里进行的吧？

2019-02-18

| 作者回复

acl_dbs是全局数组

把你使用sql语句，和语句序列发一下哦

类似按照时间顺序

session a:

xxx

xxx

session b:

xxxx

session a:

xxxx

这样

2019-02-18



发芽的紫菜

0

老师，联合索引的数据结构是怎么样的？到底是怎么存的？看了前面索引两章，还是不太懂，留言里老师说会在后面章节会讲到，但我也没看到，所以来此问一下？老师能否画图讲解一下

2019-02-18

| 作者回复

联合索引就是两个字段拼起来作索引

比如一个索引如果定义为(f1,f2),

在数据上，就是f1的值之后跟着f2的值。

查找的时候，比如执行 where f1=M and f2=N, 也是把M,N拼起来，去索引树查找

2019-02-18



晨思暮语

0

丁老师,您好：

关于上一章我留言的疑问,我重新整理了下。就是第十五章中老师留的思考题。

我模拟了老师的实验,结果有点出入,请老师帮忙看看,谢谢!

基础环境:

```
mysql> select version();
```

```
+-----+
```

```
| version() |
```

```
+-----+
```

```
| 5.7.22-log |
```

```
+-----+
```

```
1 row in set (0.00 sec)
```

```
mysql> show variables like '%tx%';
```

```
+-----+-----+
```

```
| Variable_name | Value |
```

```
+-----+-----+
```

```
| tx_isolation | REPEATABLE-READ |
```

```
| tx_read_only | OFF |
```

```
+-----+-----+
```

```
2 rows in set (0.00 sec)
```

模拟实验:

session A:

```
mysql> begin;
```

```
mysql> select * from t;
```

```
+----+----+
```

```
| id | a |
```

```
+----+----+
```

```
| 1 | 2 |
```

```
+----+----+
```

```
1 row in set (0.00 sec)
```

session B:

```
mysql> update t set a=3 where id=1;
```

```
Query OK, 1 row affected (0.00 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

SESSION A:

```
mysql> update t set a=3 where id=1;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
Rows matched: 1 Changed: 0 Warnings: 0
```

```
/*老师的实验显示为: 1 rows affected*/
```

```
mysql> select * from t where id=1;
```

```
+----+----+
```

```
| id | a |
```

```
+----+----+
```

```
| 1 | 2 |
```

```
+----+-----+
1 row in set (0.00 sec)
/*老师实验的查询结果为: 1,3 */
2019-02-18
```

| 作者回复

这个跟binlog_format有关。

如果binlog_format=row, 那么最后session A的select查到的是2;

如果binlog_format=statement, 那么最后session A的select查到的是3;

我们在文章里面有做了说明了，这个逻辑是依赖于“MySQL在执行update语句的时候，有没有把字段c也读进来”，

2019-02-26



Sinyo

0

查一张大表，order_key字段值对应的最小createtime;

以前一直用方法一查数，后来同事说可以优化成方法二，查询效率比方法一高了几倍；

mysql特有的group by功能，没有group by的字段默认取查到的第一条记录；

方法一：

```
select distinct order_key
,createtime
from (select order_key
,min(createtime) createtime
from aaa
group by order_key) a
join aaa b
on a.order_key = b.order_key
and a.createtime = b.createtime
```

方法二：

```
select order_key
,createtime
from (select order_key
,createtime
FROM aaa
order by createtime
) a
group by order_key
```

2019-02-18

| 作者回复

[]

这第二个写法跟：

`select order_key ,createtime FROM aaa force index(createtime) group by order_key`

的逻辑语义相同吗？

2019-02-18



Leon

0

老师我使用`delete`删除用户，再创建用户都是失败，但是使用`drop`就可以了

```
mysql> create user 'ua'@'%' identified by 'L1234567890c-';
```

```
ERROR 1396 (HY000): Operation CREATE USER failed for 'ua'@'%'
```

```
mysql> drop user 'ua'@'%';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> create user 'ua'@'%' identified by 'L1234567890c-';
```

```
Query OK, 0 rows affected (0.01 sec)
```

是不是`drop`才会同时从内存和磁盘删除用户信息，但是`delete`只是从磁盘删除

2019-02-18

| 作者回复

对，`drop`是同时操作磁盘和内存，

`delete`就是我们说的不规范操作

2019-02-18



爸爸回来了

0

众所周知，`sql`是不区分大小写的。然而，涉及插件的变量却不是这样；上次在配置一个插件的参数的时候，苦思良久.....最后发现了这个问题。难受

2019-02-18

| 作者回复

你说的是参数的名字，还是参数的值？

2019-02-18

43 | 要不要使用分区表？

2019-02-20 林晓斌



我经常被问到这样一个问题：分区表有什么问题，为什么公司规范不让使用分区表呢？今天，我们就来聊聊分区表的使用行为，然后再一起回答这个问题。

分区表是什么？

为了说明分区表的组织形式，我先创建一个表t：

```
CREATE TABLE `t` (
    `ftime` datetime NOT NULL,
    `c` int(11) DEFAULT NULL,
    KEY (`ftime`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
PARTITION BY RANGE (YEAR(ftime))
(PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = InnoDB,
 PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = InnoDB,
 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = InnoDB,
 PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = InnoDB);
insert into t values('2017-4-1',1),('2018-4-1',1);
```

t.frm t#P#p_2017.ibd t#P#p_2018.ibd t#P#p_2019.ibd t#P#p_others.ibd

图1 表t的磁盘文件

我在表t中初始化插入了两行记录，按照定义的分区规则，这两行记录分别落在p_2018和p_2019这两个分区上。

可以看到，这个表包含了一个.frm文件和4个.ibd文件，每个分区对应一个.ibd文件。也就是说：

- 对于引擎层来说，这是4个表；
- 对于Server层来说，这是1个表。

你可能会觉得这两句都是废话。其实不然，这两句话非常重要，可以帮助我们理解分区表的执行逻辑。

分区表的引擎层行为

我先给你举个在分区表加间隙锁的例子，目的是说明对于InnoDB来说，这是4个表。

	session A	session B
T1	begin; select * from t where ftime='2017-5-1' for update;	
T2		insert into t values('2018-2-1', 1); (Query OK) insert into t values('2017-12-1', 1); (blocked)

图2 分区表间隙锁示例

这里顺便复习一下，我在[第21篇文章](#)和你介绍的间隙锁加锁规则。

我们初始化表t的时候，只插入了两行数据，ftime的值分别是，‘2017-4-1’ 和‘2018-4-1’。

session A的select语句对索引ftime上这两个记录之间的间隙加了锁。如果是一个普通表的话，那么T1时刻，在表t的ftime索引上，间隙和加锁状态应该是图3这样的。

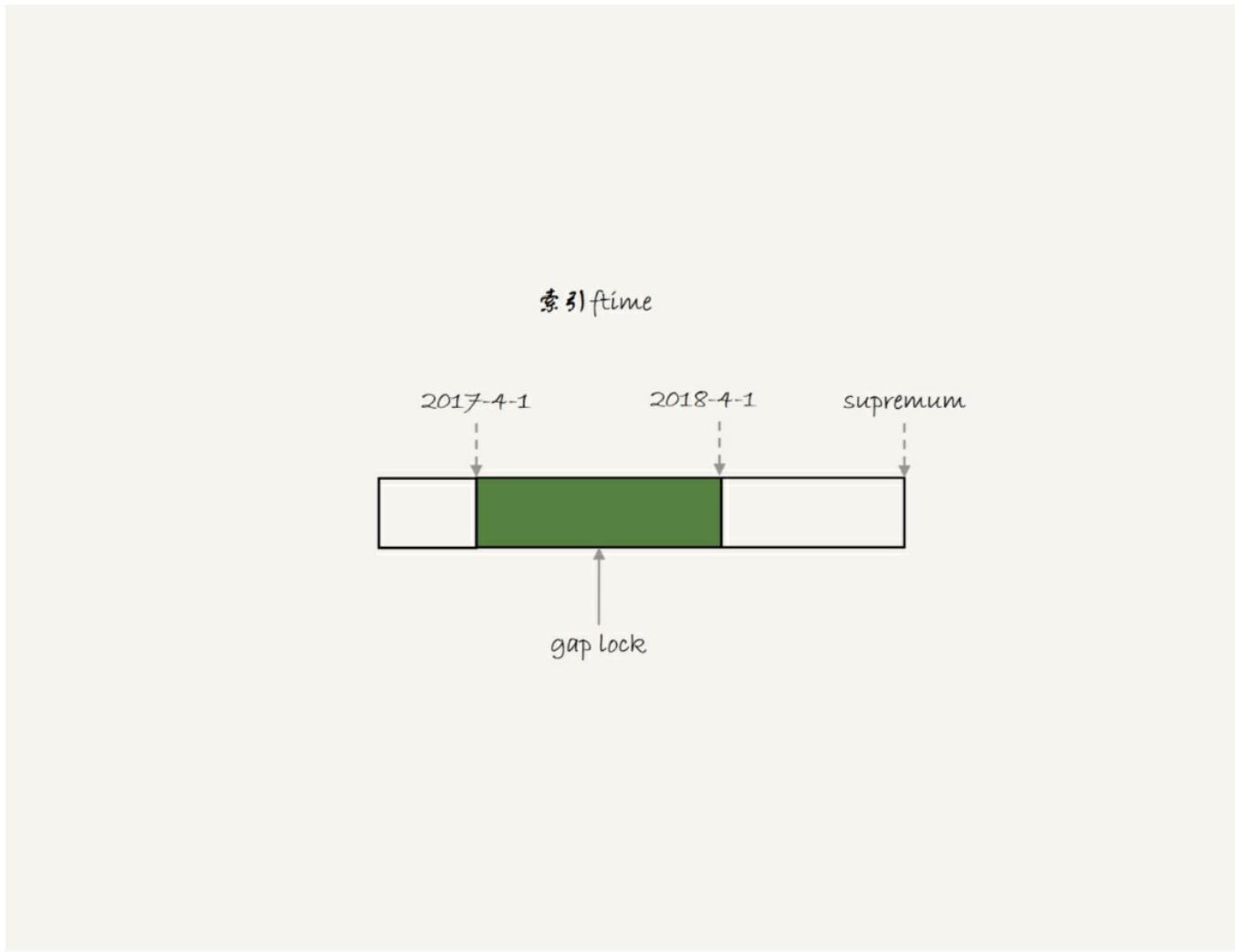


图3 普通表的加锁范围

也就是说，'2017-4-1' 和 '2018-4-1' 这两个记录之间的间隙是会被锁住的。那么，**sesion B**的两条插入语句应该都要进入锁等待状态。

但是，从上面的实验效果可以看出，**session B**的第一个insert语句是可以执行成功的。这是因为，对于引擎来说，p_2018和p_2019是两个不同的表，也就是说2017-4-1的下一个记录并不是2018-4-1，而是p_2018分区的supremum。所以T1时刻，在表t的ftime索引上，间隙和加锁的状态其实是图4这样的：

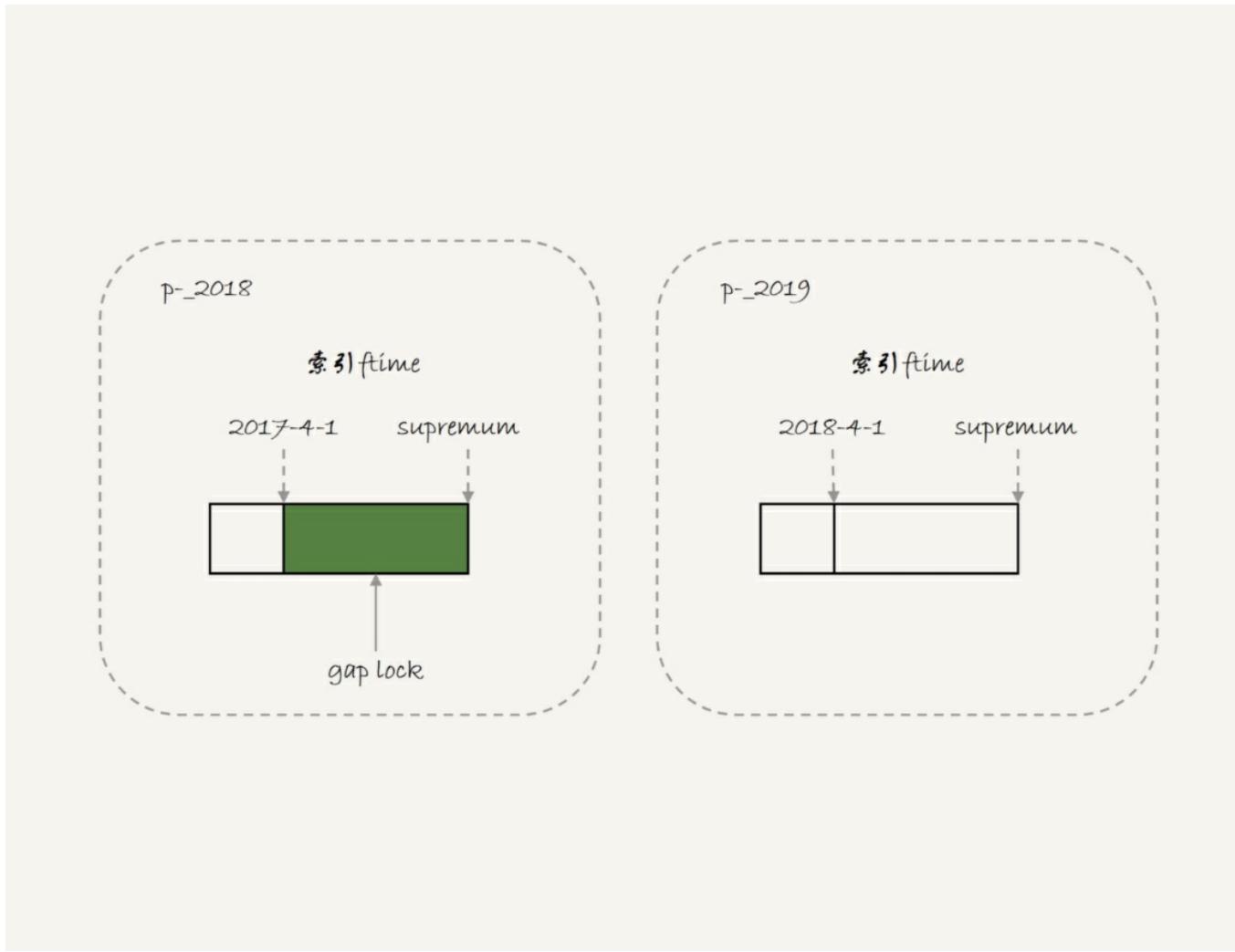


图4 分区表t的加锁范围

由于分区表的规则，**session A**的**select**语句其实只操作了分区**p_2018**，因此加锁范围就是图4中深绿色的部分。

所以，**session B**要写入一行**ftime**是2018-2-1的时候是可以成功的，而要写入2017-12-1这个记录，就要等**session A**的间隙锁。

图5就是这时候的**show engine innodb status**的部分结果。

```
----- TRX HAS BEEN WAITING 5 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 24 page no 4 n bits 72 index ftime of table `test`.`t` /* Partition `p_2018` */ trx id 1304 lock_mode X insert intention waiting
Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
 0: len 8; hex 73757072656d756d; asc supremum;;
```

图5 session B被锁住信息

看完**InnoDB**引擎的例子，我们再来一个**MyISAM**分区表的例子。

我首先用**alter table t engine=mysam**，把表**t**改成**MyISAM**表；然后，我再用下面这个例子说明，对于**MyISAM**引擎来说，这是4个表。

session A	session B
<pre>alter table t engine=myisam; update t set c=sleep(100) where ftime='2017-4-1';</pre>	<pre>select * from t where ftime='2018-4-1'; (Query OK) select * from t where ftime='2017-5-1'; (blocked)</pre>

图6 用MyISAM表锁验证

在**session A**里面，我用**sleep(100)**将这条语句的执行时间设置为100秒。由于**MyISAM**引擎只支持表锁，所以这条**update**语句会锁住整个表**t**上的读。

但我们看到的结果是，**session B**的第一条查询语句是可以正常执行的，第二条语句才进入锁等待状态。

这正是因为**MyISAM**的表锁是在引擎层实现的，**session A**加的表锁，其实是锁在分区**p_2018**上。因此，只会堵住在这个分区上执行的查询，落到其他分区的查询是不受影响的。

看到这里，你可能会说，分区表看来还不错嘛，为什么不让用呢？我们使用分区表的一个重要原因就是单表过大。那么，如果不使用分区表的话，我们就是要使用手动分表的方式。

接下来，我们一起看看手动分表和分区表有什么区别。

比如，按照年份来划分，我们就分别创建普通表**t_2017**、**t_2018**、**t_2019**等等。手工分表的逻辑，也是找到需要更新的所有分表，然后依次执行更新。在性能上，这和分区表并没有实质的差别。

分区表和手工分表，一个是由**server**层来决定使用哪个分区，一个是由应用层代码来决定使用哪个分表。因此，从引擎层看，这两种方式也是没有差别的。

其实这两个方案的区别，主要是在**server**层上。从**server**层看，我们就不得不提到分区表一个被广为诟病的问题：打开表的行为。

分区策略

每当第一次访问一个分区表的时候，**MySQL**需要把所有的分区都访问一遍。一个典型的报错情况是这样的：如果一个分区表的分区很多，比如超过了1000个，而**MySQL**启动的时候，**open_files_limit**参数使用的是默认值**1024**，那么就会在访问这个表的时候，由于需要打开所有的文件，导致打开表文件的个数超过了上限而报错。

下图就是我创建的一个包含了很多分区的表**t_myisam**，执行一条插入语句后报错的情况。

```
mysql> insert into t_myisam values('2017-4-1',1);
ERROR 1016 (HY000): Can't open file: './test/t_myisam.frm' (errno: 24 - Too many open files)
```

图 7 insert 语句报错

可以看到，这条insert语句，明显只需要访问一个分区，但语句却无法执行。

这时，你一定从表名猜到了，这个表我用的是MyISAM引擎。是的，因为使用InnoDB引擎的话，并不会出现这个问题。

MyISAM分区表使用的分区策略，我们称为通用分区策略（generic partitioning），每次访问分区都由server层控制。通用分区策略，是MySQL一开始支持分区表的时候就存在的代码，在文件管理、表管理的实现上很粗糙，因此有比较严重的性能问题。

从MySQL 5.7.9开始，InnoDB引擎引入了本地分区策略（native partitioning）。这个策略是在InnoDB内部自己管理打开分区的行为。

MySQL从5.7.17开始，将MyISAM分区表标记为即将弃用(deprecated)，意思是“从这个版本开始不建议这么使用，请使用替代方案。在将来的版本中会废弃这个功能”。

从MySQL 8.0版本开始，就不允许创建MyISAM分区表了，只允许创建已经实现了本地分区策略的引擎。目前来看，只有InnoDB和NDB这两个引擎支持了本地分区策略。

接下来，我们再看一下分区表在server层的行为。

分区表的server层行为

如果从server层看的话，一个分区表就只是一个表。

这句话是什么意思呢？接下来，我就用下面这个例子来和你说明。如图8和图9所示，分别是这个例子的操作序列和执行结果图。

session A	session B
begin; select * from t where ftime='2018-4-1';	
	alter table t truncate partition p_2017; (blocked)

图8 分区表的MDL锁

```
mysql> show processlist;
+----+----+----+----+----+----+----+----+
| Id | User | Host      | db   | Command | Time | State          | Info
+----+----+----+----+----+----+----+----+
| 4  | root | localhost:10196 | test | Sleep   | 219  | NULL           |
| 5  | root | localhost:10786 | test | Query   | 221  | Waiting for table metadata lock |
| 7  | root | localhost:12340 | test | Query   | 0    | starting       |
+----+----+----+----+----+----+----+----+
```

图9 show processlist结果

可以看到，虽然**session B**只需要操作**p_2107**这个分区，但是由于**session A**持有整个表**t**的**MDL**锁，就导致了**session B**的**alter**语句被堵住。

这也是**DBA**同学经常说的，分区表，在做**DDL**的时候，影响会更大。如果你使用的是普通分表，那么当你在**truncate**一个分表的时候，肯定不会跟另外一个分表上的查询语句，出现**MDL**锁冲突。

到这里我们小结一下：

1. **MySQL**在第一次打开分区表的时候，需要访问所有的分区；
2. 在**server**层，认为这是同一张表，因此所有分区共用同一个**MDL**锁；
3. 在引擎层，认为这是不同的表，因此**MDL**锁之后的执行过程，会根据分区表规则，只访问必要的分区。

而关于“必要的分区”的判断，就是根据**SQL**语句中的**where**条件，结合分区规则来实现的。比如我们上面的例子中，**where ftime='2018-4-1'**，根据分区规则**year**函数算出来的值是**2018**，那么就会落在**p_2019**这个分区。

但是，如果这个**where**条件改成**where ftime>='2018-4-1'**，虽然查询结果相同，但是这时候根据**where**条件，就要访问**p_2019**和**p_others**这两个分区。

如果查询语句的**where**条件中没有分区**key**，那就只能访问所有分区了。当然，这并不是分区表的问题。即使是使用业务分表的方式，**where**条件中没有使用分表的**key**，也必须访问所有的分表。

我们已经理解了分区表的概念，那么什么场景下适合使用分区表呢？

分区表的应用场景

分区表的一个显而易见的优势是对业务透明，相对于用户分表来说，使用分区表的业务代码更简洁。还有，分区表可以很方便的清理历史数据。

如果一项业务跑的时间足够长，往往就会有根据时间删除历史数据的需求。这时候，按照时间分区的分区表，就可以直接通过**alter table t drop partition ..**这个语法删掉分区，从而删掉过期的历史数据。

这个**alter table t drop partition ..**操作是直接删除分区文件，效果跟**drop**普通表类似。与使用**delete**语句删除数据相比，优势是速度快、对系统影响小。

小结

这篇文章，我主要和你介绍的是**server**层和引擎层对分区表的处理方式。我希望通过这些介绍，你能够对是否选择使用分区表，有更清晰的想法。

需要注意的是，我是以范围分区（**range**）为例和你介绍的。实际上，**MySQL**还支持**hash**分区、**list**分区等分区方法。你可以在需要用到的时候，再翻翻[手册](#)。

实际使用时，分区表跟用户分表比起来，有两个绕不开的问题：一个是第一次访问的时候需要访问所有分区，另一个是共用**MDL**锁。

因此，如果要使用分区表，就不要创建太多的分区。我见过一个用户做了按天分区策略，然后预先创建了**10**年的分区。这种情况下，访问分区表的性能自然是不好的。这里有两个问题需要注意：

1. 分区并不是越细越好。实际上，单表或者单分区的数据一千万行，只要没有特别大的索引，对于现在的硬件能力来说都已经是小表了。
2. 分区也不要提前预留太多，在使用之前预先创建即可。比如，如果是按月分区，每年年底时再把下一年度的**12**个新分区创建上即可。对于没有数据的历史分区，要及时的**drop**掉。

至于分区表的其他问题，比如查询需要跨多个分区取数据，查询性能就会比较慢，基本上就不是分区表本身的问题，而是数据量的问题或者说是使用方式的问题了。

当然，如果你的团队已经维护了成熟的分库分表中间件，用业务分表，对业务开发同学没有额外的复杂性，对**DBA**也更直观，自然是更好的。

最后，我给你留下一个思考题吧。

我们举例的表中没有用到自增主键，假设现在要创建一个自增字段**id**。**MySQL**要求分区表中的主键必须包含分区字段。如果要在表**t**的基础上做修改，你会怎么定义这个表的主键呢？为什么这么定义呢？

你可以把你的结论和分析写在留言区，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上篇文章后面还不够多，可能很多同学还没来记得看吧，我们就等后续有更多留言的时候，再补充本期的“上期问题时间”吧。

@夹心面包 提到了在**grant**的时候是支持通配符的：“_”表示一个任意字符，“%”表示任意字符串。这个技巧在一个分库分表方案里面，同一个分库上有多个**db**的时候，是挺方便的。不过我个人认为，权限赋值的时候，控制的精确性还是要优先考虑的。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「」请朋友读，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



夹心面包

5

我说下我的感想

1 经典的利用分区表的场景

1 zabbix历史数据表的改造,利用存储过程创建和改造

2 后台数据的分析汇总,比如日志数据,便于清理

这两种场景我们都在执行,我们对于分区表在业务采用的是hash 用户ID方式,不过大规模应用分区表的公司我还没遇到过

2 分区表需要注意的几点

总结下

1 由于分区表都很大,DDL耗时是非常严重的,必须考虑这个问题

2 分区表不能建立太多的分区,我曾被分享一个因为分区表分区过多导致的主从延迟问题

3 分区表的规则和分区需要预先设置好,否则后来进行修改也很麻烦

2019-02-20

作者回复

0 非常好

2019-02-20



aliang

2

老师, mysql还有一个参数是innodb_open_files, 资料上说作用是限制Innodb能打开的表的数量。它和open_files_limit之间有什么关系吗?

2019-02-21

作者回复

好问题。

在InnoDB引擎打开文件超过 innodb_open_files这个值的时候，就会关掉一些之前打开的文件。

其实我们文章中，InnoDB分区表使用了本地分区策略以后，即使分区个数大于open_files_limit，打开InnoDB分区表也不会报“打开文件过多”这个错误，就是innodb_open_files这个参数发挥的作用。

2019-02-21



怀刚

1

请教下采用“先做备库、切换、再做备库”DDL方式不支持AFTER COLUMN是因为BINLOG原因吗？

以上DDL方式会存在影响“有损”的吧？“无损”有哪些方案呢？如果备库承载读请求但又不能接受“长时间”延时

2019-03-09

作者回复

1. 对，binlog对原因

2. 如果延迟算损失，确实是有损的。备库上的读流量要先切换到主库（也就是为什么需要在低峰期做操作）

2019-03-09



权恒星

1

这个只适合单机吧？集群没法即使用innodb引擎，又支持分区表吧，只能使用中间件了。之前调研了一下，官方只有ndb cluster才支持分区表？

2019-02-20

作者回复

对这篇文章讲的是单机上的单表多分区

2019-02-20



One day

1

这次竟然只需要再读两次就能读懂，之前接触过mycat和sharding-jdbc实现分区，老师能否谈谈这方面的呢

2019-02-20

作者回复

赞两次

这个就是我们文章说的“分库分表中间件”

不过看到不少公司都会要在这基础上做点定制化

2019-02-20



于欣磊

0

阿里云的DRDS就是分库分表的中间件典型代表。自己实现了一个层Server访问层在这一层进行分库分表（对透明），然后MySQL只是相当于存储层。一些Join、负载Order by/Group by都在DRDS中间件这层完成，简单的逻辑插叙计算完对应的分库分表后下推给MySQL <https://www.aliyun.com/product/drds>

2019-02-25



1

0

老师确认下，5.7.9之后的innodb分区表，是访问第一个表时不会去打开所有的分区表了吗？

2019-02-25

| 作者回复

第一次访问的时候，要打开所有分区的

2019-02-25



启程

0

老师，你好，请教你个分区表多条件查询建索引的问题；

表A，

列a,b,c,d,e,f,g,h (其中b是datetime, a是uuid,其余是varchar)

主键索引，(b,a),按月分区

查询情况1：

where b>=? and b<=? order by b desc limit 500;

查询情况2：

where b>=? and b<=? and c in(?) order by b desc limit 500;

查询情况3：

where b>=? and b<=? and d in(?) and e in(?) order by b desc limit 500;

查询情况4：

where b>=? and b<=? and c in(?) and d in(?) and e in(?) order by b desc limit 500;

自己尝试建过不少索引，效果不是很好，请问老师，我要怎么建索引？？？

2019-02-25

| 作者回复

这个还是得看不同的语句的执行次数哈

如果从语句类型上看，可以考虑加上(b,c)、(b,d)这两个联合索引

2019-02-26



NICK

0

老师，如果用户分区，业务要做分页过滤查询怎么做才好？

2019-02-25

| 作者回复

分区表的用法跟普通表，在sql语句上是相同的。

2019-02-25



锋芒

0

老师，请问什么情况会出现间隙锁？能否专题讲一下锁呢？

2019-02-23

| 作者回复

20、21两篇看下

2019-02-23



daka

0

本期提到了ndb，了解了下，这个存储引擎高可用及读写可扩展性功能都是自带，感觉是不错，为什么很少见人使用呢？生产不可靠？

2019-02-21



helloworld.xs

0

请教个问题，一般mysql会有查询缓存，但是update操作也有缓存机制吗？使用mysql console第一次执行一个update SQL耗时明显比后面执行相同update SQL要慢，这是为什么？

2019-02-21

| 作者回复

update的话，主要应该第一次执行的时候，数据都读入到了

2019-02-21



万勇

0

老师，请问add column after column_name跟add column不指定位置，这两种性能上有区别吗？我们在add column 指定after column_name的情况很多。

2019-02-21

| 作者回复

仅仅看性能，是没什么差别的

但是建议尽量不要加after column_name，

也就是说尽量加到最后一列。

因为其实没差别，但是加在最后有以下两个好处：

1. 开始有一些分支支持快速加列，就是说如果你加在最后一列，是瞬间就能完成，而加了after column_name，就用不上这些优化（以后潜在的好处）

2. 我们在前面的文章有提到过，如果怕对线上业务造成影响，有时候是通过“先做备库、切换、再做备库”这种方式来执行ddl的，那么使用after column_name的时候用不上这种方式。

实际上列的数据是不应该有影响的，还是要形成好习惯！

2019-02-21



Q

0

老师 请问下 网站开发数据库表是myisam和innodb混合引擎 考虑管理比较麻烦 想统一成innodb
请问是否影响数据库或带来什么隐患吗？ 网站是网上商城购物类型的

2019-02-20

作者回复

应该统一成innodb

网上商城购物类型更要用InnoDB，因为MyISAM并不是crash-safe的。

测试环境改完回归下

2019-02-21



夹心面包

0

我觉得老师的问题可以提炼为 Mysql复合主键中自增长字段设置问题

复合索引可以包含一个auto_increment,但是auto_increment列必须是第一列。这样插入的话,只需要指定非自增长的列

语法 `alter table test1 change column id id int auto_increment;`

2019-02-20

作者回复

“但是auto_increment列必须是第一列”可以不是哦

2019-02-20



undefined

0

老师，有两个问题

1. 图三的间隙锁，根据“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”，不应该是 $(-\infty, 2017-4-1], (2017-4-1, 2018-4-1)$ 吗，图4左边的也应该是 $(-\infty, 2017-4-1], (2017-4-1, \text{supernum})$ ，是不是图画错了
2. 现有的一个表，一千万行的数据， InnoDB 引擎，如果以月份分区，即使有 MDL 锁和初次访问时会查询所有分区，但是综合来看，分区表的查询性能还是要比不分区好，这样理解对吗

思考题的答案

```
ALTER TABLE t
ADD COLUMN (id INT AUTO_INCREMENT ),
ADD PRIMARY KEY (id, ftime);
```

麻烦老师解答一下，谢谢老师

2019-02-20

作者回复

1. 我们语句里面是 `where ftime='2017-5-1'` 哈，不是“4-1”

2. “分区表的查询性能还是要比不分区好，这样理解对吗”，其实还是要看表的索引情况。

当然一定存在一个数量级N，把这N行分到10个分区表，比把这N行放到一个大表里面，效率高

2019-02-20



千木

0

老师您好，你在文章里面有说通用分区规则会打开所有引擎文件导致不可用，而本地分区规则应该是只打开单个引擎文件，那你不建议创建太多分区的原因是什么呢？如果是本地分区规则

，照例说是不会影响的吧，叨扰了

2019-02-20

作者回复

“本地分区规则应该是只打开单个引擎文件”，并不是哈，我在文章末尾说了，也会打开所有文件的，只是说本地分区规则有优化，比如如果文件数过多，就会淘汰之前打开的文件句柄（暂时关掉）。

所以分区太多，还是会有影响的

2019-02-20

郭江伟

0

此时主键包含自增列+分区键，原因为对innodb来说分区等于单独的表，自增字段每个分区可以插入相同的值，如果主键只有自增列无法完全保证唯一性。

测试表如下：

```
mysql> show create table t\G
Table: t
Create Table: CREATE TABLE `t` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`ftime` datetime NOT NULL,
`c` int(11) DEFAULT NULL,
PRIMARY KEY (`id`,`ftime`),
KEY `ftime` (`ftime`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
/*!50100 PARTITION BY RANGE (YEAR(ftime))
(PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = InnoDB,
PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = InnoDB,
PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = InnoDB,
PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */
1 row in set (0.00 sec)
```

```
mysql> insert into t values(1,'2017-4-1',1),(1,'2018-4-1',1);
```

```
Query OK, 2 rows affected (0.02 sec)
```

```
mysql> select * from t;
```

id	ftime	c
1	2017-04-01 00:00:00	1
1	2018-04-01 00:00:00	1

```
2 rows in set (0.00 sec)
```

```
mysql> insert into t values(null,'2017-5-1',1),(null,'2018-5-1',1);
```

```
Query OK, 2 rows affected (0.02 sec)
```

```
mysql> select * from t;
+---+-----+----+
| id | ftime | c |
+---+-----+----+
| 1 | 2017-04-01 00:00:00 | 1 |
| 2 | 2017-05-01 00:00:00 | 1 |
| 1 | 2018-04-01 00:00:00 | 1 |
| 3 | 2018-05-01 00:00:00 | 1 |
+---+-----+----+
4 rows in set (0.00 sec)
```

2019-02-20

| 作者回复



2019-02-24



wljs

0

老师我想问个问题 我们公司一个订单表有110个字段 想拆分成两个表 第一个表放经常查的字段 第二个表放不常查的 现在程序端不想改sql，数据库端来实现 当查询字段中 第一个表不存在 就去关联第二个表查出数据 db能实现不

2019-02-20

| 作者回复

用view可能可以实现部分你的需求，但是强烈不建议这么做。

业务不想修改，就好好跟他们说，毕竟这样分（常查和不常查的垂直拆分）是合理的，对读写性能都有明显的提升的。

2019-02-20

44 | 答疑文章（三）：说一说这些好问题

2019-02-22 林晓斌



这是我们专栏的最后一篇答疑文章，今天我们来说说一些好问题。

在我看来，能够帮我们扩展一个逻辑的边界的问题，就是好问题。因为通过解决这样的问题，能够加深我们对这个逻辑的理解，或者帮我们关联到另外一个知识点，进而可以帮助我们建立起自己的知识网络。

在工作中会问好问题，是一个很重要的能力。

经过这段时间的学习，从评论区的问题我可以感觉出来，紧跟课程学习的同学，对SQL语句执行性能的感觉越来越好了，提出的问题也越来越细致和精准了。

接下来，我们就一起看看同学们在评论区提到的这些好问题。在和你一起分析这些问题的时候，我会指出它们具体是在哪篇文章出现的。同时，在回答这些问题的过程中，我会假设你已经掌握了这篇文章涉及的知识。当然，如果你印象模糊了，也可以跳回文章再复习一次。

join的写法

在第35篇文章[《join语句怎么优化？》](#)中，我在介绍join执行顺序的时候，用的都是straight_join。@郭健同学在文后提出了两个问题：

1. 如果用left join的话，左边的表一定是驱动表吗？
2. 如果两个表的join包含多个条件的等值匹配，是都要写到on里面呢，还是只把一个条件写到

on里面，其他条件写到where部分？

为了同时回答这两个问题，我来构造两个表a和b：

```
create table a(f1 int, f2 int, index(f1))engine=innodb;
create table b(f1 int, f2 int)engine=innodb;
insert into a values(1,1),(2,2),(3,3),(4,4),(5,5),(6,6);
insert into b values(3,3),(4,4),(5,5),(6,6),(7,7),(8,8);
```

表a和b都有两个字段f1和f2，不同的是表a的字段f1上有索引。然后，我往两个表中都插入了6条记录，其中在表a和b中同时存在的数据有4行。

@郭健同学提到的第二个问题，其实就是下面这两种写法的区别：

```
select * from a left join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q1*/
select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q2*/
```

我把这两条语句分别记为Q1和Q2。

首先，需要说明的是，这两个left join语句的语义逻辑并不相同。我们先来看一下它们的执行结果。

```
mysql> select * from a left join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q1*/
+---+---+---+---+
| f1 | f2 | f1 | f2 |
+---+---+---+---+
| 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 |
| 6 | 6 | 6 | 6 |
| 1 | 1 | NULL | NULL |
| 2 | 2 | NULL | NULL |
+---+---+---+---+
6 rows in set (0.00 sec)

mysql> select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q2*/
+---+---+---+---+
| f1 | f2 | f1 | f2 |
+---+---+---+---+
| 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 |
| 6 | 6 | 6 | 6 |
+---+---+---+---+
4 rows in set (0.00 sec)
```

图1 两个join的查询结果

可以看到：

- 语句**Q1**返回的数据集是6行，表**a**中即使没有满足匹配条件的记录，查询结果中也会返回一行，并将表**b**的各个字段值填成**NULL**。
- 语句**Q2**返回的是4行。从逻辑上可以这么理解，最后的两行，由于表**b**中没有匹配的字段，结果集里面**b.f2**的值是空，不满足**where**部分的条件判断，因此不能作为结果集的一部分。

接下来，我们看看实际执行这两条语句时，MySQL是怎么做的。

我们先一起看看语句**Q1**的**explain**结果：

```
mysql> explain select * from a left join b on(a.f1=b.f1) and (a.f1=1);/*Q1*/
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id  | select_type | table | partitions | type | possible_keys | key   | key_len | ref    | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1   | SIMPLE      | a     | NULL       | ALL  | NULL          | NULL  | NULL    | NULL   | 6    | 100.00 | NULL
| 1   | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL  | NULL    | NULL   | 6    | 100.00 | Using where; Using join buffer (Block Nested Loop)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图2 Q1的explain结果

可以看到，这个结果符合我们的预期：

- 驱动表是表**a**，被驱动表是表**b**；
- 由于表**b**的**f1**字段上没有索引，所以使用的是**Block Nested Loop Join**（简称**BNL**）算法。

看到**BNL**算法，你就应该知道这条语句的执行流程其实是这样的：

- 把表**a**的内容读入**join_buffer**中。因为是**select ***，所以字段**f1**和**f2**都被放入**join_buffer**了。
- 顺序扫描表**b**，对于每一行数据，判断**join**条件（也就是**a.f1=b.f1 and a.f2=b.f2**）是否满足，满足条件的记录，作为结果集的一行返回。如果语句中有**where**子句，需要先判断**where**部分满足条件后，再返回。
- 表**b**扫描完成后，对于没有被匹配的表**a**的行（在这个例子中就是(1,1)、(2,2)这两行），把剩余字段补上**NULL**，再放入结果集中。

对应的流程图如下：

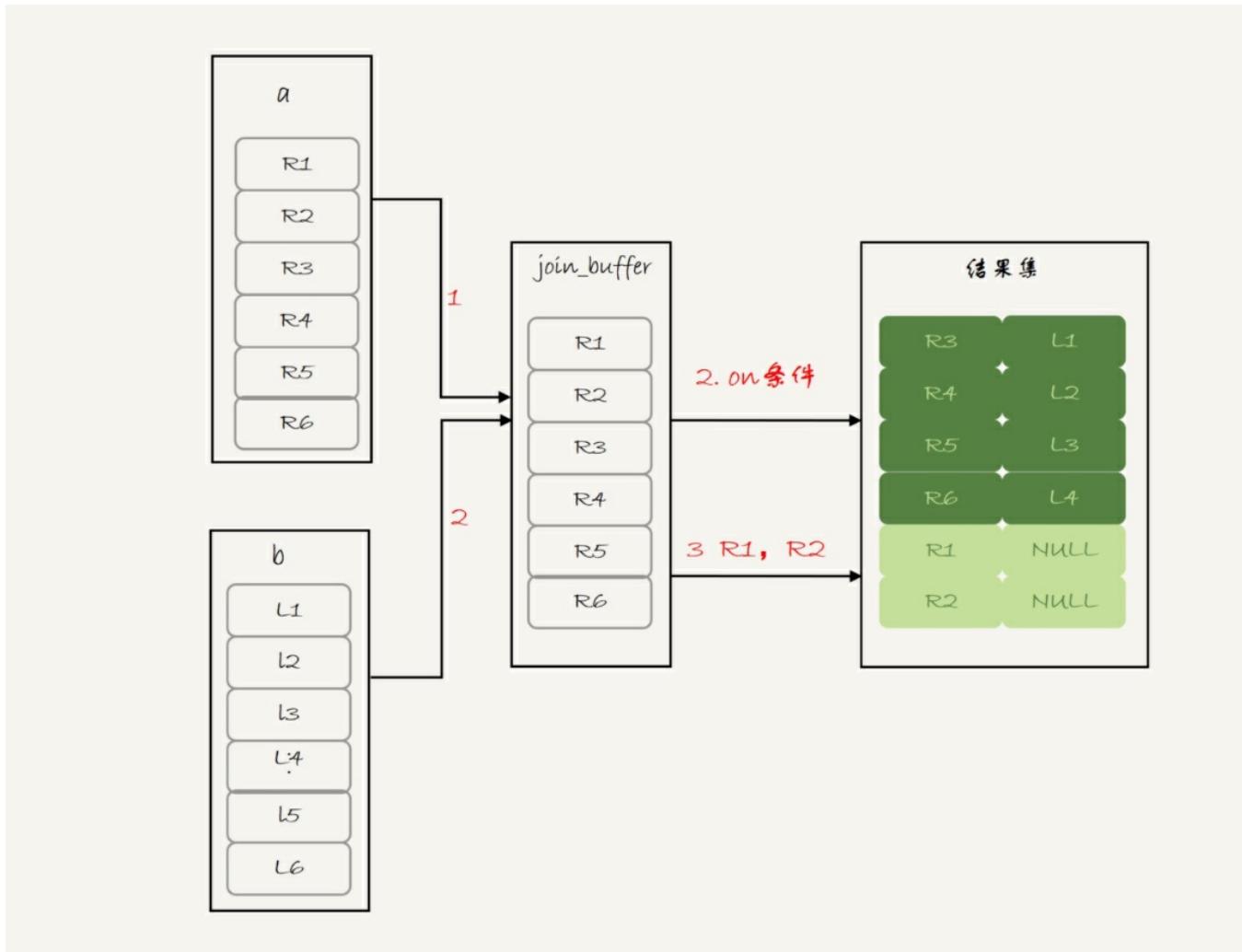


图3 left join -BNL算法

可以看到，这条语句确实是表**a**为驱动表，而且从执行效果看，也和使用**straight_join**是一样的。

你可能会想，语句**Q2**的查询结果里面少了最后两行数据，是不是就是把上面流程中的步骤3去掉呢？我们还是先看一下语句**Q2**的**explain**结果吧。

```
mysql> explain select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2); /*Q2*/
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | b    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 6   | 100.00  | Using where |
| 1 | SIMPLE     | a    | NULL       | ref  | f1            | f1  | 5       | test.b.f1 | 1   | 16.67  | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

图4 Q2的explain结果

这里先和你说一句题外话，专栏马上就结束了，我也和你一起根据**explain**结果“脑补”了很多次一条语句的执行流程了，所以我希望你已经具备了这个能力。今天，我们再一起分析一次SQL语句的**explain**结果。

可以看到，这条语句是以表**b**为驱动表的。而如果一条**join**语句的**Extra**字段什么都没写的话，就表示使用的是**Index Nested-Loop Join**（简称**NLJ**）算法。

因此，语句**Q2**的执行流程是这样的：顺序扫描表**b**，每一行用**b.f1**到表**a**中去查，匹配到记录后判断**a.f2=b.f2**是否满足，满足条件的话就作为结果集的一部分返回。

那么，为什么语句**Q1**和**Q2**这两个查询的执行流程会差距这么大呢？其实，这是因为优化器基于**Q2**这个查询的语义做了优化。

为了理解这个问题，我需要再和你交代一个背景知识点：在MySQL里，**NULL**跟任何值执行等值判断和不等值判断的结果，都是**NULL**。这里包括，**select NULL = NULL** 的结果，也是返回 **NULL**。

因此，语句**Q2**里面**where a.f2=b.f2**就表示，查询结果里面不会包含**b.f2**是**NULL**的行，这样这个**left join**的语义就是“找到这两个表里面，**f1**、**f2**对应相同的行。对于表**a**中存在，而表**b**中匹配不到的行，就放弃”。

这样，这条语句虽然用的是**left join**，但是语义跟**join**是一致的。

因此，优化器就把这条语句的**left join**改写成了**join**，然后因为表**a**的**f1**上有索引，就把表**b**作为驱动表，这样就可以用上**NLJ** 算法。在执行**explain**之后，你再执行**show warnings**，就能看到这个改写的结果，如图5所示。

```
mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message
+-----+-----+-----+
| Note | 1003 | /* select#1 */ select `test`.`a`.'f1` AS `f1`, `test`.`a`.'f2` AS `f2`, `test`.`b`.'f1` AS `f1`, `test`.`b`.'f2` AS `f2` from `test`.`a` join `test`.`b` where ((`test`.`a`.'f1` = `test`.`b`.'f1`) and (`test`.`a`.'f2` = `test`.`b`.'f2')) |
+-----+-----+-----+
```

图5 Q2的改写结果

这个例子说明，即使我们在SQL语句中写成**left join**，执行过程还是有可能不是从左到右连接的。也就是说，使用**left join**时，左边的表不一定是驱动表。

这样看来，如果需要**left join**的语义，就不能把被驱动表的字段放在**where**条件里面做等值判断或不等值判断，必须都写在**on**里面。那如果是**join**语句呢？

这时候，我们再看看这两条语句：

```
select * from a join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q3*/
select * from a join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q4*/
```

我们再使用一次看**explain** 和 **show warnings**的方法，看看优化器是怎么做的。

```

mysql> explain select * from a join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q3*/
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | b      | NULL       | ALL   | NULL          | f1  | 5        | test.b.f1 | 6    | 100.00  | Using where |
| 1 | SIMPLE     | a      | NULL       | ref   | f1            | NULL | NULL     | test.b.f1 | 1    | 16.67   | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)

mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message |
+-----+-----+-----+
| Note  | 1003 | /* select#1 */ select 'test'.`a`.f1 AS `f1`, `test`.`a`.f2 AS `f2`, `test`.`b`.f1 AS `f1`, `test`.`b`.f2 AS `f2` from `test`.`a` join `test`.`b` where ((`test`.`a`.f2 = `test`.`b`.f2) and (`test`.`a`.f1 = `test`.`b`.f1)) |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from a join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q4*/
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | b      | NULL       | ALL   | NULL          | f1  | 5        | test.b.f1 | 6    | 100.00  | Using where |
| 1 | SIMPLE     | a      | NULL       | ref   | f1            | NULL | NULL     | test.b.f1 | 1    | 16.67   | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)

mysql> show warnings;
+-----+-----+-----+
| Level | Code | Message |
+-----+-----+-----+
| Note  | 1003 | /* select#1 */ select 'test'.`a`.f1 AS `f1`, `test`.`a`.f2 AS `f2`, `test`.`b`.f1 AS `f1`, `test`.`b`.f2 AS `f2` from `test`.`a` join `test`.`b` where ((`test`.`a`.f1 = `test`.`b`.f1) and (`test`.`a`.f2 = `test`.`b`.f2)) |
+-----+-----+-----+

```

图6 join语句改写

可以看到，这两条语句都被改写成：

```
select * from a join b where (a.f1=b.f1) and (a.f2=b.f2);
```

执行计划自然也是一模一样的。

也就是说，在这种情况下，join将判断条件是否全部放在on部分就没有区别了。

Simple Nested Loop Join 的性能问题

我们知道，join语句使用不同的算法，对语句的性能影响会很大。在第34篇文章 [《到底可不可以使用join?》](#) 的评论区中，@书策稠浊 和 @朝夕心 两位同学提了一个很不错的问题。

我们在文中说到，虽然BNL算法和Simple Nested Loop Join 算法都是要判断 $M \times N$ 次（M和N分别是join的两个表的行数），但是Simple Nested Loop Join 算法的每轮判断都要走全表扫描，因此性能上BNL算法执行起来会快很多。

为了便于说明，我还是先为你简单描述一下这两个算法。

BNL算法的执行逻辑是：

1. 首先，将驱动表的数据全部读入内存join_buffer中，这里join_buffer是无序数组；
2. 然后，顺序遍历被驱动表的所有行，每一行数据都跟join_buffer中的数据进行匹配，匹配成功则作为结果集的一部分返回。

Simple Nested Loop Join 算法的执行逻辑是：顺序取出驱动表中的每一行数据，到被驱动表去做全表扫描匹配，匹配成功则作为结果集的一部分返回。

这两位同学的疑问是，**Simple Nested Loop Join** 算法，其实也是把数据读到内存里，然后按照匹配条件进行判断，为什么性能差距会这么大呢？

解释这个问题，需要用到**MySQL**中索引结构和**Buffer Pool**的相关知识点：

1. 在对被驱动表做全表扫描的时候，如果数据没有在**Buffer Pool**中，就需要等待这部分数据从磁盘读入；
从磁盘读入数据到内存中，会影响正常业务的**Buffer Pool**命中率，而且这个算法天然会对被驱动表的数据做多次访问，更容易将这些数据页放到**Buffer Pool**的头部（请参考[第35篇文章](#)中的相关内容）；
2. 即使被驱动表数据都在内存中，每次查找“下一个记录的操作”，都是类似指针操作。而在**join_buffer**中是数组，遍历的成本更低。

所以说，**BNL** 算法的性能会更好。

distinct 和 **group by** 的性能

在第37篇文章[《什么时候会使用内部临时表？》](#)中，@老杨同志 提了一个好问题：如果只需要去重，不需要执行聚合函数，**distinct** 和 **group by** 哪种效率高一些呢？

我来展开一下他的问题：如果表**t**的字段**a**上没有索引，那么下面这两条语句：

```
select a from t group by a order by null;  
select distinct a from t;
```

的性能是不是相同的？

首先需要说明的是，这种**group by**的写法，并不是**SQL**标准的写法。标准的**group by**语句，是需要在**select**部分加一个聚合函数，比如：

```
select a,count(*) from t group by a order by null;
```

这条语句的逻辑是：按照字段**a**分组，计算每组的**a**出现的次数。在这个结果里，由于做的是聚合计算，相同的**a**只出现一次。

备注：这里你可以顺便复习一下[第37篇文章](#)中关于**group by**的相关内容。

没有了**count(*)**以后，也就是不再需要执行“计算总数”的逻辑时，第一条语句的逻辑就变成是：按照字段**a**做分组，相同的**a**的值只返回一行。而这就是**distinct**的语义，所以不需要执行聚合函数时，**distinct** 和 **group by**这两条语句的语义和执行流程是相同的，因此执行性能也相同。

这两条语句的执行流程是下面这样的。

1. 创建一个临时表，临时表有一个字段**a**，并且在这个字段**a**上创建一个唯一索引；
2. 遍历表**t**，依次取数据插入临时表中：
 - 如果发现唯一键冲突，就跳过；
 - 否则插入成功；
3. 遍历完成后，将临时表作为结果集返回给客户端。

备库自增主键问题

除了性能问题，大家对细节的追问也很到位。在第39篇文章[《自增主键为什么不是连续的？》](#)评论区，@帽子掉了同学问到：在**binlog_format=statement**时，语句A先获取**id=1**，然后语句B获取**id=2**；接着语句B提交，写**binlog**，然后语句A再写**binlog**。这时候，如果**binlog**重放，是不是会发生语句B的**id**为1，而语句A的**id**为2的不一致情况呢？

首先，这个问题默认了“自增**id**的生成顺序，和**binlog**的写入顺序可能是不同的”，这个理解是正确的。

其次，这个问题限定在**statement**格式下，也是对的。因为**row**格式的**binlog**就没有这个问题了，**Write row event**里面直接写了每一行的所有字段的值。

而至于为什么不会发生不一致的情况，我们来看一下下面的这个例子。

```
create table t(id int auto_increment primary key);
insert into t values(null);
```

```
BEGIN
/*!*/;
# at 486
# at 518
#190219 18:42:49 server id 1  end_log_pos 518 CRC32 0x6364946b  Intvar
SET INSERT_ID=1/*!*/;
#190219 18:42:49 server id 1  end_log_pos 618 CRC32 0xb6277773  Query      thread_id=4      exec_time=0      error_code=0
SET TIMESTAMP=1550572969/*!*/;
insert into t values(null)
```

图7 **insert** 语句的**binlog**

可以看到，在**insert**语句之前，还有一句**SET INSERT_ID=1**。这条命令的意思是，这个线程里下一次需要用到自增值的时候，不论当前表的自增值是多少，固定用**1**这个值。

这个`SET INSERT_ID`语句是固定跟在`insert`语句之前的，比如@帽子掉了同学提到的场景，主库上语句A的id是1，语句B的id是2，但是写入`binlog`的顺序先B后A，那么`binlog`就变成：

```
SET INSERT_ID=2;
```

语句B:

```
SET INSERT_ID=1;
```

语句A:

你看，在备库上语句B用到的`INSERT_ID`依然是2，跟主库相同。

因此，即使两个`INSERT`语句在主备库的执行顺序不同，自增主键字段的值也不会不一致。

小结

今天这篇答疑文章，我选了4个好问题和你分享，并做了分析。在我看来，能够提出好问题，首先表示这些同学理解了我们文章的内容，进而又做了深入思考。有你们在认真的阅读和思考，对我来说是鼓励，也是动力。

说实话，短短的三篇答疑文章无法全部展开同学们在评论区留下的高质量问题，之后有的同学还会二刷，也会有新的同学加入，大家想到新的问题就请给我留言吧，我会继续关注评论区，和你在评论区交流。

老规矩，答疑文章也是要有课后思考题的。

在[第8篇文章](#)的评论区，@XD同学提到一个问题：他查看了一下`innodb_trx`，发现这个事务的`trx_id`是一个很大的数（281479535353408），而且似乎在同一个`session`中启动的会话得到的`trx_id`是保持不变的。当执行任何加写锁的语句后，`trx_id`都会变成一个很小的数字（118378）。

你可以通过实验验证一下，然后分析看看，事务id的分配规则是什么，以及MySQL为什么要这么设计呢？

你可以把你的结论和分析写在留言区，我会在下一篇文章和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

上期的问题是，怎么给分区表t创建自增主键。由于MySQL要求主键包含所有的分区字段，所以肯定是要创建联合主键的。

这时候就有两种可选：一种是(`ftime, id`)，另一种是(`id, ftime`)。

如果从利用率上来看，应该使用(`ftime, id`)这种模式。因为用`ftime`做分区key，说明大多数语句都

是包含`ftime`的，使用这种模式，可以利用前缀索引的规则，减少一个索引。

这时的建表语句是：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `ftime` datetime NOT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`ftime`,`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1
PARTITION BY RANGE (YEAR(ftime))
(PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = MyISAM,
 PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = MyISAM,
 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = MyISAM,
 PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = MyISAM);
```

当然，我的建议是你要尽量使用InnoDB引擎。InnoDB表要求至少有一个索引，以自增字段作为第一个字段，所以需要加一个`id`的单独索引。

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `ftime` datetime NOT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`ftime`,`id`),
  KEY `id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
PARTITION BY RANGE (YEAR(ftime))
(PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = InnoDB,
 PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = InnoDB,
 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = InnoDB,
 PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = InnoDB);
```

当然把字段反过来，创建成：

```
PRIMARY KEY (`id`,`ftime`),
KEY `id` (`ftime`)
```

也是可以的。

评论区留言点赞板：

@夹心面包、@郭江伟 同学提到了最后一种方案。

@aliang 同学提了一个好问题，关于`open_files_limit`和`innodb_open_files`的关系，我在回复中做了说明，大家可以看一下。

@万勇 提了一个好问题，实际上对于现在官方的版本，将字段加在中间还是最后，在性能上是没差别的。但是，我建议大家养成习惯（如果你是DBA就帮业务开发同学养成习惯），将字段加在最后面，因为这样还是比较方便操作的。这个问题，我也在评论的答复中做了说明，你可以看一下。

The image shows a promotional banner for a MySQL course. At the top left is the 'Geektime' logo. The main title 'MySQL 实战 45 讲' is displayed prominently in large, bold, dark font. Below it, a subtitle reads '从原理到实战，丁奇带你搞懂 MySQL' (From principle to practical application, Ding Qi will guide you to understand MySQL). To the right of the text is a portrait of the instructor, Lin Xiaobin, a man with glasses and a black shirt, with his arms crossed. At the bottom left, there's a section for new upgrades, encouraging users to invite friends to read and receive cash rewards.

新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



永恒记忆

2

老师，看评论包括您的回复说“`left join` 后加上 `where` 的话，肯定会被优化器优化成 `join where` 的形式，那是否下次写 `left join ..where` 的时候，不如直接写成 `join .. where`”，这个也是分情况的吧比如还是文章中的2张表，`select * from a left join b on(a.f1=b.f1) where (a.f2=2);/*Q5*/ and select * from a join b on(a.f1=b.f1) where (a.f2=2);/*Q6*/` 这个`left join`和`join`的语意和返回结果都不一样，怎么能直接写成`join`呢，如果是`where b.f2=xx` 的`where`条件可以直接写成`join`因为根据结果是不需要`left`的。

2019-02-25

作者回复

嗯 我的意思是，如果**where**条件里面，用到了**b.f2**的判断，干脆就直接写成**join**，不需要**left join**了

如果业务逻辑需要**left join**，就要把条件都放到**on**里面

业务逻辑正确性还是优先的

2019-02-25



还一棵树

1

看到 **BNL** 算法，你就应该知道这条语句的执行流程其实是这样

文章中的流程是写错了？还是我理解的有问题

- 1、如果是**a**表数据放入**join buffer**，根据**b**的每一条记录去判断是否在**a**中 如果在则保留记录
这个更像是**b left join a**。而不是**a left join b**
- 2、如果按照这个流程，比如**a**里面有2行重复的数据，如果拿**b**的数据在**a**中判断，存在则保留
，那结果集只有一条数据，而按照**a left join b** 会出现2条结果的

2019-02-26

作者回复

“如果按照这个流程，比如**a**里面有2行重复的数据，如果拿**b**的数据在**a**中判断，存在则保留，
那结果集只有一条数据，”

不会呀，你看它是这样的：

假设**join buffer**中有两个行1

然后被驱动表取出一个1，

跟**join buffer**中第一个1比较，发现满足条件，放到结果集；

跟**join buffer**中第二个1比较，发现满足条件，放到结果集；

是得到两行的

2019-03-01



宝玉

1

老师，**BNI**算法，如果**where**条件中有驱动表的过滤条件，也不会在**join**时候全部载入内存吧？

2019-02-25

作者回复

对，驱动表现过滤，然后进**join buffer**

2019-02-25



千木

1



老师您好，join使用join_buffer和内存区别那个问题的第一点解释我还是有些纳闷，你说由于从磁盘拿数据到内存里面会导致等等的性能问题我能够理解，但是说即使使用nbl算法也会涉及到从磁盘拿数据到内存吧，所以这点导致两种算法执行差异貌似不太合理，您觉得呢？

2019-02-23

| 作者回复

BNL算法拿的数据是确定的只会拿一次（遍历一遍）

而simple nested loop join是会遍历多次的

2019-02-24



白永伟

1

老师，关于备库自增id我有一个问题。既然binlog不管是statement模式还是row模式，里面的insert语句跟着的自增id都是固定的。那假如发生主备切换，备库变成主库后，客户端往新主库里插入数据时，自增id的起始值是多少，有没有可能跟已有的记录id冲突？尤其是备库还没有处理完同步过来的binlog就开始接受客户端请求时。如果要求备库必须处理完binlog才能接受客户端请求，那么怎么保证主备切换的过程中，不影响用户使用。谢谢。

2019-02-22

| 作者回复

“自增id的起始值是多少，有没有可能跟已有的记录id冲突？”

如果没有主备延迟就不会出现；

“尤其是备库还没有处理完同步过来的binlog就开始接受客户端请求时。”，对，这种情况就会。
。

“如果要求备库必须处理完binlog才能接受客户端请求，那么怎么保证主备切换的过程中，不影响用户使用”一般都是有这个要求的。要尽量减少影响，就是控制主备延迟。

2019-02-24



Chris

0

这两天在线上遇到一个比较诡异的事情，突然有几分钟连不上MySQL，通过error日志和监控的processlist显示，MySQL把很多链接都kill掉了，但处于sleep状态和show status的语句没有kill，看监控的资源使用情况不是很高，只是innodb rows read指标特别高，现在完全是没头绪了

2019-03-15

| 作者回复

看看是不是有什么外部工具在工作

2019-03-16



长杰

0

select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2);

老师，这个语句是否可以理解为：先按照on后面的条件关联，获取结果集，然后对结果集用where条件进行二次过滤？

2019-03-02

作者回复

要看索引哈

如果**b**上的索引只有**f1**是的，

如果**b**上的索引是(**f1,f2**)，就两个一起关联了

【咱们文中说了，这个语句会被转成普通join哦】

2019-03-02



长杰

0

把表**a**的内容读入**join_buffer**中。因为是**select ***，所以字段**f1**和**f2**都被放入**join_buffer**了。

顺序扫描表**b**，对于每一行数据，判断**join**条件（也就是**a.f1=b.f1 and a.f2=b.f2**）是否满足，满足条件的记录，作为结果集的一行返回。如果语句中有**where**子句，需要先判断**where**部分满足条件后，再返回。

表**b**扫描完成后，对于没有被匹配的表**a**的行（在这个例子中就是(1,1)、(2,2)这两行），把剩余字段补上**NULL**，再放入结果集中。

是否可以理解为：假如有**where**条件的情况下，对与满足**on**条件的行，再去过滤**where**条件，满足就返回；对于不满足**on**条件的行，**b**字段补**Null**后返回，不需要再过滤**where**条件

2019-03-02

作者回复

不是，如果有**where**，并且**where**里面有用到**b.f1**或**b.f2**，那就要求结果集里面没有这些**null**的行。

就是说**where a.f2=b.f2**的意思是

Where (a.f2 is not null) and (b.f2 is not null) and (a.f2 = b.f2)

2019-03-02



梦康

0

留言的人太多，辛苦老实答疑了。虽然我的问题没能被翻牌子

2019-02-25

作者回复

不好意思，确实你的问题比较难一些

最近在做收尾的工作，后面一定会把问题都清理掉的哈。

你得问题质量高，是我喜欢回答的问题类型

2019-02-25



PYH

0

你好 我想问一下mysql能实现oracle的拉链表么。如果能前提条件是什么？

2019-02-24



龙文

0

明白了 谢谢老师！

2019-02-24

| 作者回复

0

2019-02-24



滔滔

0

老师您好，想请问下在innodb引擎rr隔离级别下，单独的一条update语句是不是默认就是一个事务(在执行update前不输入begin)，而单独的一条select语句是不是不会开启一个事务，哪怕是"当前读"也不会开启一个事务，更进一步，是不是对表的增删改操作默认都会开启一个事务？

2019-02-24

| 作者回复

1. 单独一个update，会启动一个事务
2. 单独一个select，也会启动一个事务
3. innodb表，增删改查都会启动一个事务

2019-02-24



发条橙子。

0

啧啧，原来我写的left join一直都不是标准的，每次后面都会加上where，还一直以为左面是驱动表。既然实际上left join后加上where的话，肯定会被优化器优化成join where的形式，那是否下次写left join ..where的时候，不如直接写成join .. where，省去优化器自己去优化，这样是不是稍稍快些

2019-02-23

| 作者回复

是的

如果原来就有where，说明原来其实也不用left join

2019-02-23



龙文

0

老师你好，我在第21讲求助了一个死锁问题，当时你回复说后面会解答，不过我浏览了下后续文章没找到解答，所以再次求助下。ps:用的阿里云的rds,提了工单没效果啊

作者回复: 有的，你看一下第40篇“insert唯一键冲突”这一段

ps:我已经离开阿里云挺久的了

谢谢老师,我看了第40篇,还是有地方不太明白,再打扰下

mysql 版本5.6

隔离级别为rc

```
CREATE TABLE `uk_test` (  
    `id` bigint(20) NOT NULL AUTO_INCREMENT,
```

```
`a` int(11) NOT NULL,  
`b` int(11) NOT NULL,  
`c` int(11) NOT NULL,  
PRIMARY KEY (`id`),  
UNIQUE KEY `uk_a_b` (`a`,`b`) USING BTREE  
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8mb4
```

表中数据:

```
+----+----+----+
```

id a b c

```
+----+----+----+
```

1 1 1 2

6 1 2 1

```
+----+----+----+
```

sql:执行顺序

session1:begin;

session2:begin;

session1:select * from uk_test where a = 1 and b = 1 for update;

session2:select * from uk_test where a = 1 and b = 1 for update;

session1:insert into uk_test (a,b,c) values(1,1,2) on duplicate key update c = 2;

session2:ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
on

我的疑问是:

1.rc隔离级别下对唯一键的insert也会加next-key lock吗?

2.死锁日志显示

session 1已经成功加上行锁(lock_mode X locks rec but not gap),

session 2在等待同一个行锁(lock_mode X locks rec but not gap waiting),

session1这时因为等待lock_mode X waiting而死锁。

这里的lock_mode X waiting是指next-key lock吗?

如果是的话, 没想明白这里怎么形成资源循环等待了?

我的猜测是session1 这时持有行锁, 要next-key lock 所以要去加gap锁。session 2持有gap锁在等行锁。但如果是这样为什么session2 在rc下select for update, 且记录存在时会加gap锁?还有gap锁加锁不是不互斥吗?

2019-02-23

作者回复

1. 会

2. 你这里

session 1 成功加锁一个record lock;

session 2执行的是一个select 语句, 而且a=1 and b=1就只锁一行 (a, b上有联合唯一索引)

，这里就是要申请一个记录行锁(but not gap waiting)。

这里虽然没有加锁成功，但是已经加入了锁队列（只是这个锁是处于等待状态）

---这时候队列里面有两个锁对象了

然后**session 1**再**insert**失败的时候，就要加**next-key lock**，（注意这个锁对象跟第一个锁对象不同）。

然后死锁检测看到，2号锁在等1号锁；3号要等2号，而3和1又是同一个**session**，就认为是死锁了。

2019-02-24



龙文

0

老师你好，我在第21讲求助了一个死锁问题，当时你回复说后面会解答，不过我浏览了下后续文章没找到解答，所以再次求助下。**ps**:用的阿里云的rds,提了工单没效果啊

2019-02-23

| 作者回复

有的，你看一下第40篇“**insert**唯一键冲突”这一段

ps:我已经离开阿里云挺久的了』

2019-02-23



夜空中最亮的星 (华仔)

0

订阅了好几个专栏 **mysql**这个是最优先看的，别的专栏可以跟不上这个必须跟上，老师计划出第二季吗？

2019-02-22



夜空中最亮的星 (华仔)

0

这么快就要结束，好快啊

2019-02-22

| 作者回复

跟进得很快啊大家』

2019-02-22



一大只』

0

老师好，我做了下课后题的实验，不清楚为啥设计，下面记录了我看到的现象，不一定对哈。

使用**start transaction with consistent snapshot**；

同一个**session**的开启快照事务后的**trx_id**很大并且一致，如果关闭这个**session**，开启另一个**session**使用**snapshot**，初始值的**trx_id**也是与之前的**session**一致的。

如果再打开第二个**session**使用**snapshot**，第一次查询**trx**表，会发现第一个**session**还是很大只，第二个打开的**trx_id**会很小，但这个很小的**trx_id**是第一个打开的**session**的最小**trx_id+1**。这时，如果**commit**；再**start snapshot**，那么将会出现一个比第一个**session**还要大一点的**trx_id**，我开了几个**session**，第一次是**+24**，随后都是加**12**，如下图：

+-----+-----+

```
| trx_mysql_thread_id | trx_id |
+-----+-----+
| 14672 | 421259275839776 |
| 14661 | 421259275838864 |
| 14645 | 421259275837952 |
| 14587 | 421259275837040 |
| 14578 | 421259275835216 |
+-----+-----+
```

只有一个session打开snapshot情况下，trx_id在commit后会增加，但在事务内不会看到trx_id增加，使用select,select lock in share mode不会导致trx_id增加。

一个ddl操作应该是 trx_id+18

不在事务内的dml操作：

```
delete 1条 trx_id+2
delete 多条 trx_id+6
insert 1条 trx_id+1
insert values (),()...多条trx_id+5
update 1条 trx_id+2
update 多条 trx_id+6
```

snapshot事务内的dml操作：

事务内先select * from tb for update;再delete from tb where id=xxx;这样的delete trx_id+1
如果是事务内直接delete from tb where id=xxx;或delete from tb;这样的delete trx_id+6

事务内update 1条 trx_id+2, 如果先select * from tb for update;再update 1条, 有时候是trx_id+2
, 有时候是trx_id+5

事务内update 多条 trx_id+6

2019-02-22

作者回复

很好的验证

下一篇文章除会讲到哈

2019-02-23



万勇

0

感谢老师上一期的解答，还请教一个分区表的问题，分区表创建的聚集索引是分区本地维护的吧，但是主键索引要保证全局唯一性。那分区和主键索引之间是不是要建立一种关系？另外分区表如果我们创建普通索引，按道理可以分区创建的，分区维护自己的普通索引，各分区之间互不影响。

2019-02-22

作者回复

就是我这篇末尾建议的几种建表方法，就是建立联系了

2019-02-22



克劳德

0

老师好，如果**group by**用作数据去重，根据文章中描述的，流程2会遍历表依次插入进临时表。我理解的遍历表是通过扫描主键索引来做的，因此同一组的记录只会留下主键值最小的那个，是否正确？

能否通过扫描其他索引，来达到去重后的记录不按照主键值来决定？

2019-02-22

| 作者回复

1. 对，就是扫描这个索引的过程中，第一个碰到的值
2. 可以，你用**force index**试试

2019-02-22

45 | 自增id用完怎么办？

2019-02-25 林晓斌



MySQL里有很多自增的**id**，每个自增**id**都是定义了初始值，然后不停地往上加步长。虽然自然数是没有上限的，但是在计算机里，只要定义了表示这个数的字节长度，那它就有上限。比如，无符号整型(**unsigned int**)是4个字节，上限就是 $2^{32}-1$ 。

既然自增**id**有上限，就有可能被用完。但是，自增**id**用完了会怎么样呢？

今天这篇文章，我们就来看看**MySQL**里面的几种自增**id**，一起分析一下它们的值达到上限以后，会出现什么情况。

表定义自增值**id**

说到自增**id**，你第一个想到的应该就是表结构定义里的自增字段，也就是我在第39篇文章[《自增主键为什么不是连续的？》](#)中和你介绍过的自增主键**id**。

表定义的自增值达到上限后的逻辑是：再申请下一个**id**时，得到的值保持不变。

我们可以通过下面这个语句序列验证一下：

```
create table t(id int unsigned auto_increment primary key) auto_increment=4294967295;
insert into t values(null);
//成功插入一行 4294967295

show create table t;

/* CREATE TABLE `t` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4294967295;
*/



insert into t values(null);
//Duplicate entry '4294967295' for key 'PRIMARY'
```

可以看到，第一个`insert`语句插入数据成功后，这个表的`AUTO_INCREMENT`没有改变（还是`4294967295`），就导致了第二个`insert`语句又拿到相同的自增`id`值，再试图执行插入语句，报主键冲突错误。

$2^{32}-1$ (`4294967295`) 不是一个特别大的数，对于一个频繁插入删除数据的表来说，是可能会被用完的。因此在建表的时候你需要考察你的表是否有可能达到这个上限，如果有可能，就应该创建成8个字节的`bigint unsigned`。

InnoDB系统自增row_id

如果你创建的InnoDB表没有指定主键，那么InnoDB会给你创建一个不可见的，长度为6个字节的`row_id`。InnoDB维护了一个全局的`dict_sys.row_id`值，所有无主键的InnoDB表，每插入一行数据，都将当前的`dict_sys.row_id`值作为要插入数据的`row_id`，然后把`dict_sys.row_id`的值加1。

实际上，在代码实现时`row_id`是一个长度为8字节的无符号长整型(`bigint unsigned`)。但是，InnoDB在设计时，给`row_id`留的只是6个字节的长度，这样写到数据表中时只放了最后6个字节，所以`row_id`能写到数据表中的值，就有两个特征：

1. `row_id`写入表中的值范围，是从0到 $2^{48}-1$ ；
2. 当`dict_sys.row_id`= 2^{48} 时，如果再有插入数据的行为要来申请`row_id`，拿到以后再取最后6个字节的话就是0。

也就是说，写入表的`row_id`是从0开始到 $2^{48}-1$ 。达到上限后，下一个值就是0，然后继续循环。

当然， $2^{48}-1$ 这个值本身已经很大了，但是如果一个MySQL实例跑得足够久的话，还是可能达到这个上限的。在InnoDB逻辑里，申请到`row_id=N`后，就将这行数据写入表中；如果表中已经存

在row_id=N的行，新写入的行就会覆盖原有的行。

要验证这个结论的话，你可以通过gdb修改系统的自增row_id来实现。注意，用gdb改变量这个操作是为了便于我们发现问题，只能在测试环境使用。

```
mysql> create table t(a int)engine=innodb;
gdb -p <pid of mysql> -ex 'p dict_sys.row_id=1' --batch
mysql> inset into t values(1);
gdb -p <pid.mysql> -ex 'p dict_sys.row_id=281474976710656' --batch
mysql> inset into t values(2);
mysql> inset into t values(3);
mysql> select * from t;
```

图1 row_id用完的验证序列

```
mysql> select * from t;
+---+
| a |
+---+
| 2 | row_id=0
| 3 | row_id=1
+---+
2 rows in set (0.00 sec)
```

图2 row_id用完的效果验证

可以看到，在我用gdb将dict_sys.row_id设置为 2^{48} 之后，再插入的a=2的行会出现在表t的第一行，因为这个值的row_id=0。之后再插入的a=3的行，由于row_id=1，就覆盖了之前a=1的行，因为a=1这一行的row_id也是1。

从这个角度看，我们还是应该在InnoDB表中主动创建自增主键。因为，表自增id到达上限后，再插入数据时报主键冲突错误，是更能被接受的。

毕竟覆盖数据，就意味着数据丢失，影响的是数据可靠性；报主键冲突，是插入失败，影响的是可用性。而一般情况下，可靠性优先于可用性。

Xid

在第15篇文章[《答疑文章（一）：日志和索引相关问题》](#)中，我和你介绍redo log和binlog相配合的时候，提到了它们有一个共同的字段叫作Xid。它在MySQL中是用来对应事务的。

那么，Xid在MySQL内部是怎么生成的呢？

MySQL内部维护了一个全局变量global_query_id，每次执行语句的时候将它赋值给Query_id，然后给这个变量加1。如果当前语句是这个事务执行的第一条语句，那么MySQL还会同时把Query_id赋值给这个事务的Xid。

而global_query_id是一个纯内存变量，重启之后就清零了。所以你就知道了，在同一个数据库实例中，不同事务的Xid也是有可能相同的。

但是MySQL重启之后会重新生成新的binlog文件，这就保证了，同一个binlog文件里，Xid一定是一样的。

虽然MySQL重启不会导致同一个binlog里面出现两个相同的Xid，但是如果global_query_id达到上限后，就会继续从0开始计数。从理论上讲，还是会出现同一个binlog里面出现相同Xid的场景。

因为global_query_id定义的长度是8个字节，这个自增值的上限是 $2^{64}-1$ 。要出现这种情况，必须是下面这样的过程：

1. 执行一个事务，假设Xid是A；
2. 接下来执行 2^{64} 次查询语句，让global_query_id回到A；
3. 再启动一个事务，这个事务的Xid也是A。

不过， 2^{64} 这个值太大了，大到你可以认为这个可能性只会存在于理论上。

Innodb trx_id

Xid和InnoDB的trx_id是两个容易混淆的概念。

Xid是由server层维护的。InnoDB内部使用Xid，就是为了能够在InnoDB事务和server之间做关联。但是，InnoDB自己的trx_id，是另外维护的。

其实，你应该非常熟悉这个trx_id。它就是在我们在第8篇文章[《事务到底是隔离的还是不隔离的？》](#)中讲事务可见性时，用到的事务id（transaction id）。

InnoDB内部维护了一个max_trx_id全局变量，每次需要申请一个新的trx_id时，就获得max_trx_id的当前值，然后并将max_trx_id加1。

InnoDB数据可见性的核心思想是：每一行数据都记录了更新它的trx_id，当一个事务读到一行数

据的时候，判断这个数据是否可见的方法，就是通过事务的一致性视图与这行数据的做对比。

对于正在执行的事务，你可以从information_schema.innodb_trx表中看到事务的。

我在上一篇文章的末尾留给你的思考题，就是关于从innodb trx表里面查到的的。现在，我们一起来看一个事务现场：

	session A	session B
T1	begin; select * from t limit 1;	
T2		<pre>use information_schema; select trx_id, trx_mysql_thread_id from innodb_trx; /* +-----+-----+ trx_id trx_mysql_thread_id +-----+-----+ 421578461423440 5 +-----+-----+ */</pre>
T3	insert into t values(null);	
T4		<pre>select trx_id, trx_mysql_thread_id from innodb_trx; /* +-----+-----+ trx_id trx_mysql_thread_id +-----+-----+ 1289 5 +-----+-----+ */</pre>

图3 事务的

session B里，我从innodb_trx表里查出的这两个字段，第二个字段就是线程id。显示线程id，是为了说明这两次查询看到的事务对应的线程id都是5，也就是session A所在的线程。

可以看到，T2时刻显示的是一个很大的数；T4时刻显示的是1289，看上去是一个比较正常的数字。这是什么原因呢？

实际上，在T1时刻，session A还没有涉及到更新，是一个只读事务。而对于只读事务，InnoDB并不会分配。也就是说：

1. 在T1时刻，的值其实就是0。而这个很大的数，只是显示用的。一会儿我会再和你说说

这个数据的生成逻辑。

2. 直到**session A**在T3时刻执行**insert**语句的时候, **InnoDB**才真正分配了**trx_id**。所以, T4时刻, **session B**查到的这个**trx_id**的值就是1289。

需要注意的是, 除了显而易见的修改类语句外, 如果在**select**语句后面加上**for update**, 这个事务也不是只读事务。

在上一篇文章的评论区, 有同学提出, 实验的时候发现不止加1。这是因为:

1. **update**和**delete**语句除了事务本身, 还涉及到标记删除旧数据, 也就是要把数据放到**purge**队列里等待后续物理删除, 这个操作也会把**max_trx_id+1**, 因此在一个事务中至少加2;
2. **InnoDB**的后台操作, 比如表的索引信息统计这类操作, 也是会启动内部事务的, 因此你可能看到, **trx_id**值并不是按照加1递增的。

那么, **T2**时刻查到的这个很大的数字是怎么来的呢?

其实, 这个数字是每次查询的时候由系统临时计算出来的。它的算法是: 把当前事务的**trx**变量的指针地址转成整数, 再加上 2^{48} 。使用这个算法, 就可以保证以下两点:

1. 因为同一个只读事务在执行期间, 它的指针地址是不会变的, 所以不论是在**innodb_trx**还是在**innodb_locks**表里, 同一个只读事务查出来的**trx_id**就会是一样的。
2. 如果有并行的多个只读事务, 每个事务的**trx**变量的指针地址肯定不同。这样, 不同的并发只读事务, 查出来的**trx_id**就是不同的。

那么, 为什么还要再加上 2^{48} 呢?

在显示值里面加上 2^{48} , 目的是要保证只读事务显示的**trx_id**值比较大, 正常情况下就会区别于读写事务的**id**。但是, **trx_id**跟**row_id**的逻辑类似, 定义长度也是8个字节。因此, 在理论上还是可能出现一个读写事务与一个只读事务显示的**trx_id**相同的情况。不过这个概率很低, 并且也没有什么实质危害, 可以不管它。

另一个问题是, 只读事务不分配**trx_id**, 有什么好处呢?

- 一个好处是, 这样做可以减小事务视图里面活跃事务数组的大小。因为当前正在运行的只读事务, 是不影响数据的可见性判断的。所以, 在创建事务的一致性视图时, **InnoDB**就只需要拷贝读写事务的**trx_id**。
- 另一个好处是, 可以减少**trx_id**的申请次数。在**InnoDB**里, 即使你只是执行一个普通的**select**语句, 在执行过程中, 也是要对应一个只读事务的。所以只读事务优化后, 普通的查询语句不需要申请**trx_id**, 就大大减少了并发事务申请**trx_id**的锁冲突。

由于只读事务不分配，一个自然而然的结果就是的增加速度变慢了。

但是，`max_trx_id`会持久化存储，重启也不会重置为0，那么从理论上讲，只要一个MySQL服务跑得足够久，就可能出现`max_trx_id`达到 $2^{48}-1$ 的上限，然后从0开始的情况。

当达到这个状态后，MySQL就会持续出现一个脏读的bug，我们来复现一下这个bug。

首先我们需要把当前的`max_trx_id`先修改成 $2^{48}-1$ 。注意：这个case里使用的是可重复读隔离级别。具体的操作流程如下：

```
mysql> create table t(id int primary key, c int)engine=innodb;
mysql> insert into t values(1,1);
gdb -p <pid.mysql> -ex 'p trx_sys->max_trx_id=281474976710655' --batch
```

	session A	session B
T1	begin; select * from t; // TA /* +----+----+ id c +----+----+ 1 1 +----+----+ */	
T2		update t set c=2 where id=1; begin; update t set c=3 where id=1;
T3	select * from t; /* +----+----+ id c +----+----+ 1 3 +----+----+ 脏读 */	

图 4 复现脏读

由于我们已经把系统的`max_trx_id`设置成了 $2^{48}-1$ ，所以在**session A**启动的事务**TA**的低水位就是

$2^{48}-1$ 。

在T2时刻，**session B**执行第一条**update**语句的事务**id**就是 $2^{48}-1$ ，而第二条**update**语句的事务**id**就是0了，这条**update**语句执行后生成的数据版本上的**trx_id**就是0。

在T3时刻，**session A**执行**select**语句的时候，判断可见性发现，**c=3**这个数据版本的**trx_id**，小于事务**TA**的低水位，因此认为这个数据可见。

但，这个是脏读。

由于低水位值会持续增加，而事务**id**从0开始计数，就导致了系统在这个时刻之后，所有的查询都会出现脏读的。

并且，**MySQL**重启时**max_trx_id**也不会清0，也就是说重启**MySQL**，这个**bug**仍然存在。

那么，这个**bug**也是只存在于理论上吗？

假设一个**MySQL**实例的**TPS**是每秒50万，持续这个压力的话，在17.8年后，就会出现这种情况。如果**TPS**更高，这个年限自然也就更短了。但是，从**MySQL**的真正开始流行到现在，恐怕都还没有实例跑到过这个上限。不过，这个**bug**是只要**MySQL**实例服务时间够长，就会必然出现的。

当然，这个例子更现实的意义是，可以加深我们对低水位和数据可见性的理解。你也可以借此机会再回顾下第8篇文章[《事务到底是隔离的还是不隔离的？》](#)中的相关内容。

thread_id

接下来，我们再看看线程**id**（**thread_id**）。其实，线程**id**才是**MySQL**中最常见的一种自增**id**。平时我们在查各种现场的时候，**show processlist**里面的第一列，就是**thread_id**。

thread_id的逻辑很好理解：系统保存了一个全局变量**thread_id_counter**，每新建一个连接，就将**thread_id_counter**赋值给这个新连接的线程变量。

thread_id_counter定义的大小是4个字节，因此达到 $2^{32}-1$ 后，它就会重置为0，然后继续增加。但是，你不会在**show processlist**里看到两个相同的**thread_id**。

这，是因为**MySQL**设计了一个唯一数组的逻辑，给新线程分配**thread_id**的时候，逻辑代码是这样的：

```
do {
    new_id = thread_id_counter++;
} while (!thread_ids.insert_unique(new_id).second);
```

这个代码逻辑简单而且实现优雅，相信你一看就能明白。

小结

今天这篇文章，我给你介绍了MySQL不同的自增id达到上限以后的行为。数据库系统作为一个可能需要7*24小时全年无休的服务，考虑这些边界是非常有必要的。

每种自增id有各自的应用场景，在达到上限后的表现也不同：

1. 表的自增id达到上限后，再申请时它的值就不会改变，进而导致继续插入数据时报主键冲突的错误。
2. row_id达到上限后，则会归0再重新递增，如果出现相同的row_id，后写的数据会覆盖之前的数据。
3. Xid只需要不在同一个binlog文件中出现重复值即可。虽然理论上会出现重复值，但是概率极小，可以忽略不计。
4. InnoDB的max_trx_id 递增值每次MySQL重启都会被保存起来，所以我们文章中提到的脏读的例子就是一个必现的bug，好在留给我们的时间还很充裕。
5. thread_id是我们使用中最常见的，而且也是处理得最好的一个自增id逻辑了。

当然，在MySQL里还有别的自增id，比如table_id、binlog文件序号等，就留给你去验证和探索了。

不同的自增id有不同的上限值，上限值的大小取决于声明的类型长度。而我们专栏声明的上限id就是45，所以今天这篇文章也是我们的最后一篇技术文章了。

既然没有下一个id了，课后也就没有思考题了。今天，我们换一个轻松的话题，请你来说说，读完专栏以后有什么感想吧。

这个“感想”，既可以是你读完专栏前后对某一些知识点的理解发生的变化，也可以是你积累的学习专栏文章的好方法，当然也可以是吐槽或者对未来的期望。

欢迎你给我留言，我们在评论区见，也欢迎你把这篇文章分享给更多的朋友一起阅读。



MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌

网名丁奇
前阿里资深技术专家



新版升级：点击「」，10位好友免费读，邀请订阅更有**现金奖励**。

精选留言



Continue

12

跟着学了三个多月，受益匪浅，学到了很多新的知识和其中的原理！

2019-02-25

| 作者回复

早

2019-02-25



克劳德

7

本人服务端工程师，在学习这门课之前数据库一直是我的短板，曾听朋友说MySQL或数据库中涉及了很多方面的知识点，每一个拿出来展开讲几乎都能出一本书了，对数据库是越来越忌惮，同时也因为工作上并没有过多接触，水平便一直停留在编写简单SQL层面。

在面试中被问到数据库问题，只能无奈的说这块不太清楚，也曾在网上自学过，但网上的文章知识点比较零散，很多都是给出一些结论性的观点，由于不了解其内部原理，记忆很难深刻。老实说，当初报这门课的时候就像买技术书籍一样，我相信大家都有这样的体会，以为买到了就等于学到了，所以有一段时间没有点开看过，以至于后面开始学的时候都是在追赶老师和大家的进度，唯一遗憾的地方就是没能跟老师及时留言互动。

这门课虽然是文字授课，但字里行间给我的感觉就是很亲切很舒服，为什么呢，因为老师可以把晦涩的知识变得通俗易懂，有时我在思考，如果让我来讲一个自己擅长的领域是否也能做到这一点，如果要做到的话需要什么样的知识储备呢。

最后真要感谢老师的这门课，让我从心里不再惧怕数据库问题，不管是工作还是面试中信心倍增，现在时不时都敢和我们DBA“切磋切磋”了，哈哈。

祝好~

2019-02-25

| 作者回复

『“切磋切磋”

留言不会“过时”哈，在对应的章节下面提出相关的问题，我会持续关注评论区

2019-02-25



三胖

3

老师，我才学了四分之一的课程，但是这门课已经更新完了，我是直接跑到最后一节技术篇来留言的！很想知道，后来者比如我在学到后面的课程时遇到问题留言，老师还会看会回复吗？
(老师的课程超值！！)

2019-02-25

| 作者回复

会看的

后台系统是按照留言时间显示的

而且我在这事情上有强迫症，一定会让“未处理问题”变成0的！

只是说如果是其他同学评论区问过的问题，我可能就不会重复回复了

2019-02-25



某、人

2

很遗憾没能坚持到最后，但是也很庆幸能遇到这么好的专栏。以前了解mysql都是一些零散的知识点，通过学习完专栏，不论是mysql整体架构还是基础的知识点，都有了更深的认识。以后就把老师的文档当官方文档查，出现问题先来看看专栏。

感触特别深的是，老师对于提到的每一个问题，都会严谨又认真的去回答，尽量帮助每一位同学都能有所收获。要做到这一点，是特别耗费精力的。

感谢老师的传道授业解惑，希望以后有机会能当面向老师请教问题。期待老师下一部杰作

2019-02-26

| 作者回复

刚过完年都是很忙的，找时间补上哈，等你的评论区留言^_^

2019-02-26



夜空中最亮的星 (华仔)

2

不知道是最后一篇，否则的话就慢些读完了；
我是一名运维，公司也没有DBA，所以MySQL库也归我收拾；
读了老师的专栏，操作起数据库来，心情更好了；
老师的课，让我有了想看完《高性能MySQL》的兴趣；
听了老师的课，开发都来问我数据库的问题了，高兴；
老师你会有返场吗？我猜会！
可否透漏下接下来的安排，会有续集吗？进阶吗？
不想这一别就是一生。

您的从未谋面的学生。

2019-02-25

| 作者回复

谢谢你

“开发都来问我数据库的问题了”，当年我也是这么开始“入坑”，加油

2019-02-25



极客时间

1

通过这个专栏的系统学习，梳理很多知识点、扩展了我对MySQL的认识及以后使用。感谢老师的谆谆教导！

2019-02-25



NoDBA

1

低版本`thread_id`超过 $2^{32}-1$ 后，在`general log`显示是负数，高版本貌似没有这个问题，是否高版本的`thread_id`是8字节呢？

2019-02-27

| 作者回复

主要不是定义的问题，而是打印的时候代码问题，按照这个代码输出的：

`"%5ld ", (long) thread_id`

是个bug，超过 2^{31} 就变成负数了，

新版本改了

好问题！

2019-02-28



kun

1

感觉戛然而止哈 没学够，后面还会再回顾，老师辛苦！

2019-02-26



亮

1

老师，sql 的`where`里 `< 10001` 和 `<= 10000`有什么区别吗？

2019-02-25

| 作者回复

这要看你关注的是什么

你这么问，应该这个字段是整型吧？

从查询结果可能是一样的，

不过锁的范围不同，你可以看下[21篇](#)

2019-02-25



IceGeek17

1



QUESTION /



感谢老师，课程受益匪浅，

课程结束后，如果有问题，是继续在这里的评论区提问，还是会有另外一条答疑通道？

另外，在第35篇我提了几个问题，老师还没有回答，我这里再贴一下，老师看一下

问题一：

对于BKA算法的流程理解，用文中的例子，先把t1表（小表）中查询需要的字段放入join_buffer，然后把join_buffer里的字段值批量传给t2表，先根据索引a查到id，然后得到一批主键id，再根据主键id排序，然后再根据排完序的id去主键索引查数据（这里用到MRR）

理解是否正确？

这里对于主键id排序是在哪里做的，是在join_buffer里，还是另外再开辟一块临时内存？如果在join_buffer里，那join_buffer里的每行内容是不是：t2.id + t1查询必须的字段，并且join_buffer里是根据id排序的？

问题二：

虽然MySQL官方没有支持hash join，但是之前看到文章说，MariaDB已经支持hash join，能不能后续在答疑文章中简单总结下mariaDB支持的join算法

问题三：

在实际项目中，一个比较困惑的问题，看到过这样的类似写法：

`select xxx from t1 join t2 on t1.id = t2.id for update` （目的是获取几个表上最新的数据，并且加上锁，防止数据被更新）

这里有几个问题：

1) 像这样 `join + for update`，表上的加锁规则是怎么样的？是不是在需要join的两个表上根据具体的查询执行过程都加上锁？

2) 像这样 `join + for update` 的用法是否合理？碰到这样的场景，应该怎么做？

问题四：

看过阿里输出的开发手册里，强调“最多不超过三表join”，实际项目中，给我感觉很难做到所有业务都不超过三表join，那这里的问题就是，有什么相关的经验方法，可以尽量降低参与join的数据表？

比如，在数据表里添加冗余字段，可以降低参与join的数据表数量，还有什么其他好的方法？

2019-02-25

| 作者回复

就在我们评论区，提跟文章相关的内容，会继续关注。

问题一、前面的过程理解正确，MRR过程用的是read_rnd_buffer

问题二、其实我们文中最后那个过程，你把他设想成在MySQL内部执行。。

问题三、这种复杂的语句，你要把我们两部分知识点连起来看。一个原则：`for update`的话，执行语句过程中扫到的间隙和记录都要加锁。当然最好是不这么做，拆成两个语句会好些。

问题四、还是我文中的建议，如果都用NLJ或BKA算法的join其实还好，所以看看explain。

降低join表数量的方法，基本上行就是冗余字段和拆成多个语句这两个方向了

2019-02-25



Leon

1

跟着老师终于学到了最后，每天的地铁时间无比充实，我对mysql的基本原理和工作流程大致有了初步的了解，而不是以前的增删查改，打算以后抽时间再二刷三刷，等全部搞懂后，再去看看高性能mysql这本书，如果时间允许，打算再去自己参照教程实现一个简易的DB，课程虽然结束了，仍然感觉意犹未尽，希望老师拉一个倍洽群，大家一起在里面讨论和学习

2019-02-25

| 作者回复

0

评论区一直会开放

大家到对应的文章去提相关问题

二刷三刷我也一直在哦

2019-02-25



Dkey

1

当前系统并无其他事务存在时，启动一个只读事务时（意味没有事务id），它的低高水位是怎么样的老师。

2019-02-25

| 作者回复

假设当前没有其他事务存在，假设当前的max_trx_id=N，

这时候启动一个只读事务，它的高低水位就都是N。

2019-02-25



shawn

1

受益匪浅，最后几讲还想了解下null值如何建立索引，由于null直接不能比较和排序，MySQL能区分出每一个null值吗

2019-02-25

| 作者回复

可以，因为普通索引上都有主键值对吧，

所以其实是 (null, id1), (null, id2)

2019-02-25



亢星东

0

id是有上限的，这个的id上限是45，这个结局可以，讲的不错，学到很多

2019-03-13



Bamboo

0

今天终于读完了，从对MySQL只停留在CRUD操作的水平，慢慢开始对MySQL底层的机制有了一些认识，在遇到问题时，会首先从底层原理去分析，并结合explain来验证自己的分析，一次很nice的学习之旅。感谢大神老师这么认真负责，节假日都不休息，哈哈！

2019-03-12

| 作者回复



2019-03-13



ArtistLu

0

相遇恨晚，安慰下自己，种树的最好时机是十年前，其次是现在！！！谢谢老师

2019-03-08

| 作者回复



2019-03-09



fighting

0

已经二刷了，准备三刷四刷

2019-03-07

| 作者回复



2019-03-09



沙漠里的骆驼

0

讲的非常好，是我遇到课程讲授最好的了。今天刚和池老师说，希望可以有线下的课程，比如完成一个数据库的完整设计，从最上层的sql语法解析器到底层的文件调度系统。在集中的时间里面比如1个月或者2个月，线下组织大家一起，每个人都完成一个tiny_db的工程。我想这是最好的成长了。不知道老师是否也有这方面的想法？

不管如何，真的很感谢老师。如此娓娓道来，所谓的如沐春风便是如此吧。

2019-03-06

| 作者回复

谢谢你。

后面只要还是在评论区继续和大家交流

2019-03-07



芬

0

学习到了很多平时没有关注到的小细节，很赞！当然 师傅领进门 修行靠个人。剩下的就是自己好好消化应用了，谢谢老师

2019-02-28



封建的风

0

之前很多知识点有点粗浅，尤其在行版本可见性，redo log&bin log关系，加锁的原理章节，深入浅出，受益匪浅。感谢老师精品专栏，后期再二刷

2019-02-27

结束语 | 点线网面，一起构建MySQL知识网络

2019-02-27 林晓斌



林晓斌 (网名丁奇)
前阿里资深技术专家

你好，我是林晓斌。

我们一起度过了**108**天，学习了**46**篇文章，
阅读了**220,377**字，收听了近**12**个小时的音频。

点线网面，一起构建 MySQL 知识网络。



极客时间

时光流逝，这是专栏的最后一篇文章。回顾整个过程，如果用一个词来描述，就是“没料到”：

我没料到文章这么难写，似乎每一篇文章都要用尽所学；

我没料到评论这么精彩，以致于我花在评论区的时间并不比正文少；

我没料到收获这么大，每一次被评论区的提问问到盲点，都会带着久违的兴奋去分析代码。

如果让我自己评价这个专栏：

我最满意的部分，是每一篇文章都带上了实践案例，也尽量讲清楚了原理；

我最得意的段落，是在讲事务隔离级别的时候，把文章重写到第三遍，终于能够写上“到这里，我们把一致性读、当前读和行锁就串起来了”；

我最开心的时候，是看到评论区有同学在回答课后思考题时，准确地用上了之前文章介绍的知识点。因为我理解的构建知识网络，就是这么从点到线，从线到网，从网到面的过程，很欣喜能跟大家一起走过这个过程。

当然，我更看重的还是你的评价。所以，当我看到你们在评论区和知乎说“好”的时候，就只会更细致地设计文章内容和课后思考题。

同时，我知道专栏的订阅用户中，有刚刚接触MySQL的新人，也有使用MySQL多年的同学。所

以，我始终都在告诫自己，要尽量让大家都能有所收获。

在我的理解里，介绍数据库的文章需要有操作性，每一个操作有相应的原理，每一个原理背后又有它的原理，这是一个链条。能够讲清楚链条中的一个环节，就可能是一篇好文章。但是，每一层都有不同的受众。所以，我给这45篇文章定的目标就是：讲清楚操作和第一层的原理，并适当触及第二层原理。希望这样的设计不会让你觉得太浅。

有同学在问MySQL的学习路径，我在这里就和你谈谈我的理解。

1. 路径千万条，实践第一条

如果你问一个DBA“理解得最深刻的知识点”，他很可能告诉你是他踩得最深的那个坑。由此，“实践”的重要性可见一斑。

以前我带新人的时候，第一步就是要求他们手动搭建一套主备复制结构。并且，平时碰到问题的时候，我要求要动手复现。

从专栏评论区的留言可以看出来，有不少同学在跟着专栏中的案例做实验，我觉得这是个非常好的习惯，希望你能继续坚持下去。在阅读其他技术文章、图书的时候，也是同样的道理。如果你觉得自己理解了一个知识点，也一定要尝试设计一个例子来验证它。

同时，在设计案例的时候，我建议你也设计一个对照的反例，从而达到知识融汇贯通的目的。就像我在写这个专栏的过程中，就感觉自己也涨了不少知识，主要就得益于给文章设计案例的过程。

2. 原理说不清，双手白费劲

不论是先实践再搞清楚原理去解释，还是先明白原理再通过实践去验证，都不失为一种好的学习方法，因人而异。但是，怎么证明自己是不是真的把原理弄清楚了呢？答案是说出来、写出来。

如果有人请教你某个知识点，那真是太好了，一定要跟他讲明白。不要觉得这是在浪费时间。因为这样做，一来可以帮你验证自己确实搞懂了这个知识点；二来可以提升自己的技术表达能力，毕竟你终究要面临和这样的三类人讲清楚原理的情况，即：老板、晋升答辩的评委、新工作的面试官。

我在带新人的时候，如果这一届的新人不止一个，就会让他们组成学习小组，并定期给他们出一个已经有确定答案的问题。大家分头去研究，之后在小组内进行讨论。如果你能碰到愿意跟你结成学习小组的同学，一定要好好珍惜。

而“写出来”又是一个更高的境界。因为，你在写的过程中，就会发现这个“明白”很可能只是一个假象。所以，在专栏下面写下自己对本章知识点的理解，也是一个不错的夯实学习成果的方法。

3. 知识没体系，转身就忘记

把知识点“写下来”，还有一个好处，就是你会发现这个知识点的关联知识点。深究下去，点就连成线，然后再跟别的线找交叉。

比如，我们专栏里面讲到对临时表的操作不记录日志，然后你就可以给自己一个问题，这会不会导致备库同步出错？再比如，了解了临时表在不同的**binlog**格式下的行为，再追问一句，如果创建表的时候是**statement**格式，之后再修改为**row**格式（或者反之），会怎么样呢？

把这些都搞明白以后，你就能够把临时表、日志格式、同步机制，甚至于事务机制都连起来了。

相信你和我一样，在学习过程中最喜欢的就是这种交叉的瞬间。交叉多了，就形成了网络。而有了网络以后，吸收新知识的速度就很快了。

比如，如果你对事务隔离级别弄得很清楚了，在看到第45篇文章讲的**max_trx_id**超限会导致持续脏读的时候，相信你理解起来就很容易了。

4. 手册补全面，案例扫盲点

有同学还问我，要不要一开始就看手册？我的建议是不要。看手册的时机，应该是你的知识网络构建得差不多的时候。

那你可能会问，什么时候算是差不多呢？其实，这没有一个固定的标准。但是，有一些基本实践可以帮你去做一个检验。

- 能否解释清楚错误日志（**error log**）、慢查询日志（**slow log**）中每一行的意思？
- 能否快速评估出一个表结构或者一条**SQL**语句，设计得是否合理？
- 能否通过**explain**的结果，来“脑补”整个执行过程（我们已经在专栏中练习几次了）？
- 到网络上找**MySQL**的实践建议，对于每一条做一次分析：
 - 如果觉得不合理，能否给出自己的意见？
 - 如果觉得合理，能否给出自己的解释？

那，怎么判断自己的意见或者解释对不对呢？最快速、有效的途径，就是找有经验的人讨论。比如说，留言到我们专栏的相关文章的评论区，就是一个可行的方法。

这些实践做完后，你就应该对自己比较有信心了。这时候，你可以再去看手册，把知识网络中的盲点补全，进而形成面。而补全的方法就是前两点了，理论加实践。

我希望这45篇文章，可以在你构建**MySQL**知识体系的过程中，起到一个加速器的作用。

我特意安排在最后一篇文章，和你介绍**MySQL**里各种自增**id**达到定义的上限以后的不同行为。“45”就是我们这个专栏的**id**上限，而这一篇结束语，便是超过上限后的第一个值。这是一个未定义的值，由你来定义：

- 有的同学可能会像表定义的自增**id**一样，就让它定格在这里；

- 有的同学可能会像**row_id**一样，二刷，然后用新的、更全面的理解去替代之前的理解；
- 也许最多的情况是会像**thread_id**一样，将已经彻底掌握的文章标记起来，专门刷那些之前看过、但是已经印象模糊的文章。

不论是哪一种策略，只要这45篇文章中，有那么几个知识点，像**Xid**或者**InnoDB trx_id**一样，持久化到了你的知识网络里，你和我在这里花费的时间，就是“极客”的时间，就值了。

这是专栏的最后一篇文章的最后一句话，江湖再见。



林晓斌 (网名丁奇)
前阿里资深技术专家

“

不知道在学习过程中，你有哪些体会和评价？
这里有一份专栏调查问卷，邀请你填写。

**在3月4日前提交，
极客时间赠送给你专属优惠券。**

我们一起继续成长！

去提交

精选留言

 独家记忆
老师辛苦了

2019-02-27

25

 Stalary
感谢老师，是我学的最棒的课了，每次老师都会认真回答我的问题，谢谢～

2019-02-27

16

| 作者回复

谢谢你提的好问题！

2019-02-27



HuaMax

13

不说再见，让我们再建一张表，从0开始！

2019-02-27

| 作者回复

哈哈，再建一张表

如果有下个系列，我一定用这个做开头！

2019-02-27



憶海拾貝

11

这是我在极客时间第一篇跟着从开始到结束的专栏,进度偶尔落下也抓紧时间补上.

正是丁奇老师的用心，才有我们的收获良多.

辛苦啦！

2019-02-27

| 作者回复

也谢谢你的支持。这个留言时间...确认了是真爱！

大家有收获我是真开心

2019-02-27



allean

6

江湖路远，有缘再见

2019-02-27

| 作者回复

江湖路远，有缘再见！

2019-02-27



星期六男爵

5

值得二刷

2019-02-27



18岁的马化腾

4

我：喂！

老师：怎么了

我：去哪啊

老师：回家

我：然后呢

老师：上班啊

.....

我：不上班行不行

老师：不上班你养我啊

我：喂！

老师：又怎么了
我：我订阅你啊
老师：先把这期钱付了再说死鬼

2019-03-01

| 作者回复

手动惊讶[]

2019-03-01



冷笑的花猫

点赞 3

感谢老师，一期不落的学习了，虽然有很多不理解的地方，但会努力的二刷，三刷去弄懂他。以后有问题在留言板留言的时候，希望老师能百忙之中抽空回答下，最后再次感谢老师。

2019-02-27

| 作者回复

会的会的，评论区继续开放，也感谢你们一路陪伴^_^

2019-02-27



感谢老师，这门课对我的帮助真的很大，作为一个开发新人以后遇到mysql的问题的时候，将会有更多维度和方面去思考问题的本质。妈妈再也不用担心我的mysql啦，感谢老师，谢谢。

点赞 2

2019-02-27

| 作者回复

“将会有更多维度和方面去思考问题的本质”

III

2019-02-27



chenming886

点赞 2

终于跟随着林老师步伐走过了这四十多讲！让我每天地铁的行程不在漫长！谢谢

2019-02-27

| 作者回复

[]

想起来我以前在北京上班的时候，每天坐40分钟地铁[]

2019-02-27



kanxiaojie

点赞 2

目前为止，收益最多的一门课了，很多知识点已经在业务开发中实现了，而且收到的效果很好。点赞！！！

2019-02-27

| 作者回复

很多知识点已经在业务开发中实现了，而且收到的效果很好。

最高评价，开心 O(∩_∩)O

2019-02-27



南友力max先森

极客时间最好专栏

1

2019-03-09



Geek_515b9e

1

对比其他的课程，老师真的很用心。。

2019-03-04



ryp

1

老师讲得太棒了。本人2月12日发现了老师的专栏然后上的车，每天使劲刷，现在终于刷完了。本人做java开发，用mysql快2年了。之前虽然会写各种很复杂的sql，但是对底层原理并不清楚。经过这十几天的学习，本人充分领略到了mysql的各种底层原理和深坑。感谢老师！一周后，本人定会再次回来复习一遍。

2019-03-02

| 作者回复

|| 平均一天两篇，很有拼劲||

2019-03-02



萤火虫

1

林老师别走 还没听够

2019-02-28



燃烧的M豆

1

老师牛逼。。。从没给别人推荐过课程 看完根本忍不住想安利

2019-02-27



poppy

1

老师，我想请教下，对于字符串类型的索引，它的B+树是如何组织的呢？就是对于一个字符串在树节点中，如果判断大小关系呢？

2019-02-27

| 作者回复

字符串跟整数一样的呀

判断规则就是字符串规则 "az" < "ba" < "bc"

2019-02-27



强哥

1

文章质量非常高，没有深厚功底写不出，非常感谢作者的付出，自己收获很多，文章值得反复阅读及思考。期待老师再创佳作！

2019-02-27



Magic

1

感谢老师！我虽然才阅读了前十四篇文章，但受益匪浅，老师的文章除丝剥茧，图文并茂，层次合理，对学生的疑问回答耐心细致。我想老师平时工作也很繁忙，这些工作一定占据了您绝大部分休息时间。衷心感谢老师，自己也要以老师为榜样，踏踏实实把技术琢磨透

2019-02-27

| 作者回复

赞踏踏实实把技术琢磨透

2019-02-27



akuan

1

我是个MySQL小白，但每次感觉林老师标题选的图片都很有代表性

2019-02-27

| 作者回复

这个要感谢美女编辑，图片都她选的 1

2019-02-27

直播回顾 | 林晓斌：我的 MySQL 心路历程

2018-12-18 林晓斌



在专栏上线后的11月21日，我来到极客时间做了一场直播，主题就是“我的MySQL心路历程”。今天，我特意将这个直播的回顾文章，放在了专栏下面，希望你可以从我这些年和MySQL打交道的经历中，找到对你有所帮助的点。

这里，我先和你说一下，在这个直播中，我主要分享的内容：

1. 我和MySQL打交道的经历；
2. 你为什么要了解数据库原理；
3. 我建议的MySQL学习路径；
4. DBA的修炼之道。

我的经历

以丰富的经历进入百度

我是福州大学毕业的，据我了解，那时候我们学校的应届生很难直接进入百度，都要考到浙江大学读个研究生才行。没想到的是，我投递了简历后居然进了面试。

入职以后，我跑去问当时的面试官，为什么我的简历可以通过筛选？他们说：“因为你的简历厚啊”。我在读书的时候，确实做了很多项目，也实习过不少公司，所以简历里面的经历就显得很

丰富了。

在面试的时候，有个让我印象很深刻的事儿。面试官问我说，你有这么多实习经历，有没有什么比较好玩儿的事？我想了想答道，跟你说个数据量很大的事儿，在跟移动做日志分析的时候我碰到了几千万行的数据。他听完以后就笑了。

后来，我进了百度才知道，几千万行那都是小数据。

开始尝试看源码解决问题

加入百度后，我是在贴吧做后端程序，比如权限系统等等。其实很简单，就是写一个C语言程序，响应客户端请求，然后返回结果。

那个时候，我还仅仅是个MySQL的普通用户，使用了一段时间后就出现了问题：一个跑得很快的请求，偶尔会又跑得非常慢。老板问这是什么原因，而我又不好意思说不知道，于是就自己上网查资料。

但是，2008年那会儿，网上资料很少，花了挺长时间也没查出个所以然。最终，我只好去看源码。翻到源码，我当时就觉得它还蛮有意思的。而且，源码真的可以帮我解决一些问题。

于是一发不可收拾，我从那时候就入了源码的“坑”。

混社区分享经验

2010年的时候，阿里正好在招数据库的开发人员。虽然那时我还只是看得懂源码，没有什么开发经验，但还是抱着试试看的态度投了简历。然后顺利通过了面试，成功进入了阿里。之后，我就跟着褚霸（霸爷）干了7年多才离开了阿里。

在百度的时候，我基本上没有参加过社区活动。因为那时候百度可能更提倡内部分享，解决问题的经验基本上都是在内网分享。所以，去了阿里以后，我才建了博客、开了微博。我在阿里的花名叫丁奇，博客、微博、社区也因此都是用的这个名字。

为什么要了解数据库原理？

这里，我讲几个亲身经历的事情，和你聊聊为什么要了解数据库原理。

了解原理能帮你更好地定位问题

一次同学聚会，大家谈起了技术问题。一个在政府里的同学说，他们的系统很奇怪，每天早上都得重启一下应用程序，否则就提示连接数据库失败，他们都不知道该怎么办。

我分析说，按照这个错误提示，应该就是连接时间过长了，断开了连接。数据库默认的超时时间是8小时，而你们平时六点下班，下班之后系统就没有人用了，等到第二天早上九点甚至十点才上班，这中间的时间已经超过10个小时了，数据库的连接肯定就会断开了。

我当时说，估计这个系统程序写得比较差，连接失败也不会重连，仍然用原来断掉的连接，所以就报错了。然后，我让他回去把超时时间改得长一点。后来他跟我说，按照这个方法，问题已经解决了。

由此，我也更深刻地体会到，作为开发人员，即使我们只知道每个参数的意思，可能就可以给出一些问题的正确应对方法。

了解原理能让你更巧妙地解决问题

我在做贴吧系统的时候，每次访问页面都要请求一次权限。所以，这个请求权限的请求，访问概率会非常高，不可能每次都去数据库里查，怎么办呢？

我想了个简单的方案：在应用程序里面开了个很大的内存，启动的时候就把整张表全部load到内存里去。这样再有权限请求的时候，直接从内存里取就行了。

数据库重启时，我的进程也会跟着重启，接下来就会到数据表里面做全表扫描，把整个用户相关信息全部塞到内存里面去。

但是，后来我遇到了一个很郁闷的情况。有时候MySQL崩溃了，我的程序重新加载权限到内存里，结果这个select语句要执行30分钟左右。本来MySQL正常重启一下是很快的，进程重启也很快，正常加载权限的过程只需要两分钟就跑完了。但是，为什么异常重启的时候就要30分钟呢？

我没辙了，只好去看源码。然后，我发现MySQL有个机制，当它觉得系统空闲时会尽量去刷脏页。

具体到我们的例子里，MySQL重启以后，会执行我的进程做全表扫描，但是因为这个时候权限数据还没有初始化完成，我的Server层不可能提供服务，于是MySQL里面就只有我那一个select全表扫描的请求，MySQL就认为现在很闲，开始拼命地刷脏页，结果就吃掉了大量的磁盘资源，导致我的全表扫描也跑得很慢。

知道了这个机制以后，我就写了个脚本，每隔0.5秒发一个请求，执行一个简单的SQL查询，告诉数据库其实我现在很忙，脏页刷得慢一点。

脚本一发布使用，脏页果然刷得慢了，加载权限的扫描也跑得很快了。据说我离职两年多以后，这个脚本还在用。

你看，如果我们懂得一些参数，并可以理解这些参数，就可以做正确的设置了。而如果我们进一步地懂得一些原理，就可以更巧妙地解决问题了。

看得懂源码让你有更多的方法

2012年的时候，阿里双十一业务的压力比较大。当时还没有这么多的SSD，是机械硬盘的时

代。

为了应对压力我们开始引入SSD，但是不敢把SSD直接当存储用，而是作为二级缓存。当时，我们用了一个叫作**Flashcache**的开源系统（现在已经是老古董级别了，不知道你有没有听过这个系统）。

Flashcache实现，把SSD当作物理盘的二级缓存，可以提升性能。但是，我们自己部署后发现性能提升的效果没有预想的那么好，甚至还不如纯机械盘。

于是，我跟霸爷就开始研究。霸爷负责分析**Flashcache**的源码，我负责分析**MySQL**源码。后来我们发现**Flashcache**是有脏页比例的，当脏页比例到了80%就会停下来强行刷盘。

一开始我们认为这个脏页比例是全部的20%，看了源码才知道，原来它分了很多个桶，比如说一个桶20M，这个桶如果用完80%，它就认为脏页满了，就开始刷脏页。这也就意味着，如果你是顺序写的话，很容易就会把一个桶写满。

知道了这个原理以后，我就把日志之类顺序写的数据全都放到了机械硬盘，把随机写的数据放到了**Flashcache**上。这样修改以后，效果就好了。

你看，如果能看得懂源码，你的操作行为就会不一样。

MySQL学习路径

说到**MySQL**的学习路径，其实我上面分享的这些内容，都可以归结为学习路径。

首先你要会用，要去了解每个参数的意义，这样你的运维行为（使用行为）就会不一样。千万不要从网上拿了一些使用建议，别人怎么用，你就怎么用，而不去想为什么。再往后，就要去了解每个参数的实现原理。一旦你了解了这些原理，你的操作行为就会不一样。再进一步，如果看得懂源码，那么你对数据库的理解也会不一样。

再来讲讲我是怎么带应届生的。实践是很好的学习方式，所以我会让新人来了以后先搭主备，然后你就会发现每个人的自学能力都不一样。比如遇到有延迟，或者我们故意构造一个主备数据不一致的场景，让新人了解怎么分析问题，解决问题。

如果一定要总结出一条学习路径的话，那首先要会用，然后可以发现问题。

在专栏里面，我在每篇文章末尾，都会提出一个常见问题，作为思考题。这些问题都不会很难，是跟专栏文章挂钩、又是会经常遇到的，但又无法直接从文章里拿到答案。

我的建议是，你可以尝试先不看答案自己去思考，或者去数据库里面翻一翻，这将会是一个不错的过

再下一步就是实践。之后当你觉得开始有一些“线”的概念了，再去**MySQL**的官方手册。在我

的专栏里，有人曾问我不要直接去看手册？

我的建议是，一开始千万不要着急看手册，这里面有100多万个英文单词，你就算再厉害，也是看了后面忘了前面。所以，你一定要自己先有脉络，然后有一个知识网络，再看手册去查漏补缺。

我自己就是这么一路走过来的。

另外，在专栏的留言区，很多用户都希望我能推荐一本书搭配专栏学习。如果只推荐一本的话，我建议你读一下《高性能MySQL》这本书，它是MySQL这个领域的经典图书，已经出到第三版了，你可以想象一下它的流行度。

这本书的其中两位译者（彭立勋、翟卫祥）是我原团队的小伙伴，有着非常丰富的MySQL源码开发经验，他们对MySQL的深刻理解，让这本书保持了跟原作英文版同样高的质量。

极客时间的编辑说，他们已经和出版社沟通，为我们专栏的用户争取到了全网最低价，仅限3天，你可以直接[点击链接](#)购买。

数据库中的
倚天屠龙

原价：¥103

优惠价 **¥86.9** 限时3天

12.22日恢复原价

高性能 MySQL

DBA的修炼

DBA和开发工程师有什么相同点？

我带过开发团队，也带过DBA团队，所以可以分享一下这两个岗位的交集。

其实，DBA本身要有些开发底子，比如说做运维系统的开发。另外，自动化程度越高，DBA的日常运维工作量就越少，DBA得去了解开发业务逻辑，往业务架构师这个方向去做。

开发工程师也是一样，不能所有的问题都指望DBA来解决。因为，DBA在每个公司都是很少的几个人。所以，开发也需要对数据库原理有一定的了解，这样向DBA请教问题时才能更专业，更高效地解决问题。

所以说，这两个岗位应该有一定程度的融合，即：开发要了解数据库原理，DBA要了解业务和

开发。

DBA有前途吗？

这里我要强调的是，每个岗位都有前途，只需要根据时代变迁稍微调整一下方向。

像原来开玩笑说DBA要体力好，因为得搬服务器。后来DBA的核心技能成了会搭库、会主备切换，但是现在这些也不够用了，因为已经有了自动化系统。

所以，DBA接下来一方面是要了解业务，做业务的架构师；另一方面，是要有前瞻性，做主动诊断系统，把每个业务的问题挑出来做成月报，让业务开发去优化，有不清楚的地方，开发同学会来找你咨询。你帮助他们做好了优化之后，可以把优化的指标呈现出来。这将很好地体现出你对于公司的价值。

有哪些比较好的习惯和提高SQL效率的方法？

这个方法，总结起来就是：要多写SQL，培养自己对SQL语句执行效率的感觉。以后再写或者建索引的时候，知道这个语句执行下去大概的时间复杂度，是全表扫描还是索引扫描、是不是需要回表，在心里都有一个大概的概念。

这样每次写出来的SQL都会快一点，而且不容易犯低级错误。这也正式我开设这个专栏的目标。

看源码需要什么技术？

看源码的话，一是要掌握C和C++；另外还要熟悉一些调试工具。因为代码是静态的，运行起来是动态的，看代码是单线程的，运行起来是多线程的，所以要学会调试。

另外，我不建议你用可视化的工具。虽然可视化工具很方便，但你不知道这个操作点下去以后，实际上做了什么，所以我建议你自己手写代码和SQL语句，这样对一些底层原理你会更明白。

怎么学习C、C++？

我在读研究生的时候，在C和C++语言的学习上进步最大。

那时，我去给专科上C和C++的课。我觉得自己已经会了，完全可以教得了。但去了之后，我才知道，自己会跟能够教别人完全是两码事儿。备课的时候，你不能只讲会用的部分，还得把原理讲清楚。这样，就会倒逼自己进行更深入更全面的学习。

有的人看完技术博客和专栏，会把这篇文章的提纲列一下，写写自己的问题和对这篇文章的理解。这个过程，是非常利于学习的。因为你听进来是一回事儿，讲出去则是另一回事儿。

学数据库要保持什么心态？

不只是数据库，所有多线程的服务，调试和追查问题的过程都是很枯燥的，遇到问题都会很麻烦。但是，你找出问题时的那一下会很爽。

我觉得你得找到这种感觉，它可以支持你度过接下来要枯燥很久的那段时光，这样你才能继续坚持下去。

当然，如果有更快乐的学习过程还是更好的，希望这个专栏能让你学习得轻松些。

极客时间

MySQL 实战 45 讲

从原理到实战，丁奇带你搞懂 MySQL

林晓斌 网名丁奇
前阿里资深技术专家



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。