08 | 事务到底是隔离的还是不隔离的?

2018-11-30 林晓斌

08 | 事务到底是隔离的还是不隔离...

朗读人：林晓斌 14′17″ | 13.09M

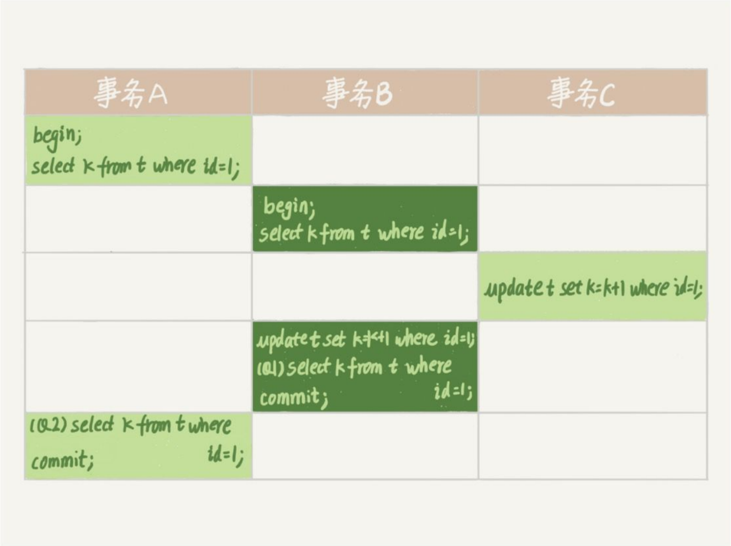
我在第 3 篇文章和你讲事务隔离级别的时候提到过，如果是可重复读隔离级别，事务 T 启动的时候

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，这个事务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，一个事务如果要更新一行，而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
begin; select k from t where id=1;		
	begin; select k from t where id=1;	
		update t set k=k+1 where id=1;
	update t set k=k+1 where id=1; (Q1) select k from t where id=1; commit;	
(Q2) select k from t where id=1; commit;		

图 1 事务 A、B、C 的执行流程

这里需要特别注意的是，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 的。

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。语句 Q1 在事务 B 中，更新了行之后查询；Q2 在只读事务 A

中查询，并且时间顺序上是在 Q1 的后面。

这时，如果我告诉你语句 Q1 返回的 k 的值是 3，而语句 Q2 返回的 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，可重复度）隔离级别的实现。

  它没有物理结构，用来在事务执行期间定义“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。

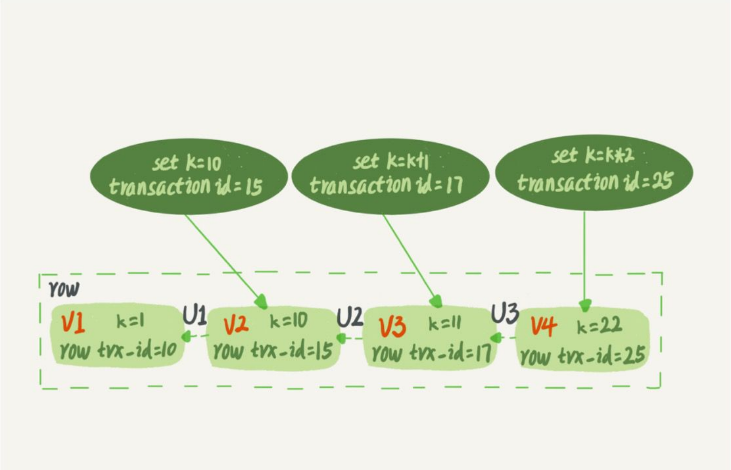


图 2 行状态变更图

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 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，InnoDB 代码实现上，一个事务只需要在启动的时候，找到所有已经提交的事务 ID 的最大值，记为 up_limit_id；然后声明说，“如果一个数据版本的 row trx_id 大于 up_limit_id，我就不认，我必须要找到它的上一个版本”。当然，如果一个事务自己更新的数据，它自己还是要认的。

> 备注：up_limit_id 来源于源码里面的变量名，我没有想到更好的名字来称呼它。

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，产生的新的数据版本的 row trx_id 都会大于 up_limit_id，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。

比如，对于图 2 中的数据来说，如果有一个事务，

它的 up_limit_id 是 18，那么当它访问这一行数据时，就会从 V4 通过 U3 算出 V3，在它看来，这一行的值是 11。

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 Q2 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，系统里面已经提交的事务最大 ID 是 99；
2. 事务 A、B、C 的版本号分别是 100、101、102，且当前系统里没有别的事务；
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、B、C 的 up_limit_id 的值就都是 99。

为了简化分析，我先把其他干扰语句去掉，只画出了跟 Q2 查询逻辑有关的操作。

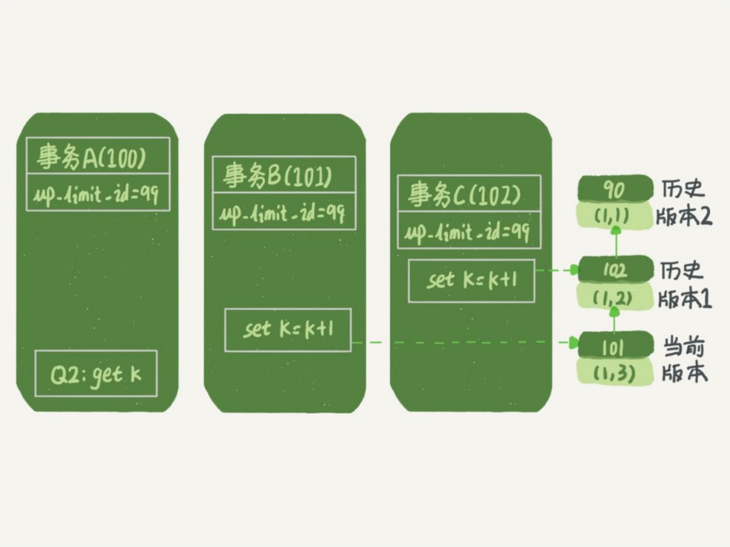


图 3 Q2 数据逻辑图

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。

好，现在事务 A 要来读数据了，它的 up_limit_id 是 99。当然了，读数据都是从当前版本读起的。所以，Q2 的读数据流程是这样的：

- 找到 (1,3) 的时候，判断出 row trx_id=101 大于 up_limit_id，要不起；
- 接着，找到上一个历史版本，一看 row trx_id=102，还是要不起；
- 再往前找，终于找到了（1,1)，它的 row trx_id=90，是可以承认的数据。

这样执行下来，事务 A 读到的这个数据，跟它在刚开始启动的时候读到的相同，所以我们称之为一致性读。

这里你可以顺便再想一个问题。**(1,1) 这个历史版本，什么时候可以被删除掉呢？**

答案是，当没有事务再需要它的时候，就可以删

掉。

如果只考虑图 1 中的三个事务的话，事务 B 只需要访问到 (1,3) 就可以，而事务 C 需要访问到的是 (1,2)。也就是说，在事务 A 提交后，(1,1) 这个版本就可以被删掉了。

更新逻辑

细心的同学可能有疑问了：**事务 B 的 update 语句，读的到底是哪个版本？**这里，我给你画了一个只看事务 B、C 的状态图。

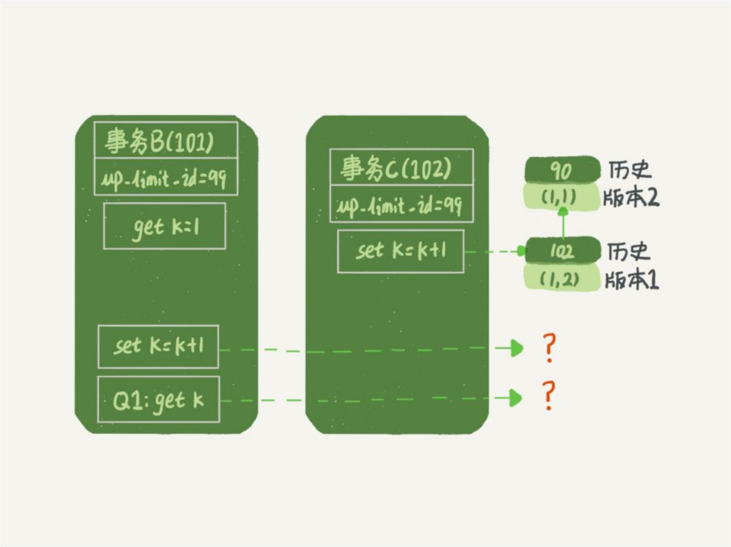


图 4 Q1 数据逻辑图

这个状态，就是事务 B 刚要执行更新时的状态。

事务 B 前面的查询语句，拿到的 k 也是 1。但是，当它要去更新数据的时候，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，当前读取到的数据是 (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 的 Q1 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，可以用，所以 Q1 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 Q2 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，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 sha

mysql> select k from t where id=1 for update;
```

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找到那个 up_limit_id，之后事务里的其他查询都共用这个 up_limit_id；
- 在读提交隔离级别下，每一个语句执行前都会重新算一次 up_limit_id 的值。

那么，我们再看一下，在读提交隔离级别下，语句 Q1 和 Q2 返回的 k 的值，分别应该是多少呢？

下面是读提交时的状态图， 可以看到 Q1、Q2 语句的 up_limit_id 发生了变化。

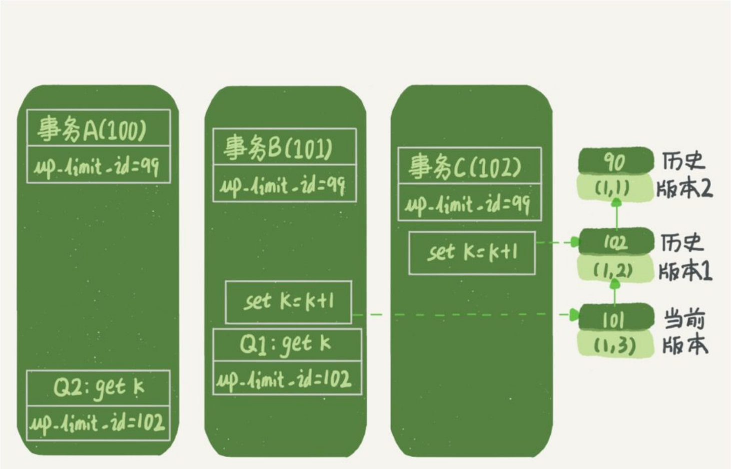


图 5 读提交隔离级别下的事务状态图

这时，事务 A 的 Q2 语句开始执行的时候，由于事务 B（101）、C（102）都已经提交了，所以 Q2 的 up_limit_id 的值就应该是事务 C 的 transaction id，即 102。那么，它在读到（1,3）的时候，就满足了 up_limt_id(102) ⩾row trx_id(101) 的条件，所以返回了 k=3。

显然地，语句 Q1 的查询结果 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，每个事务或者语句有自己的 up_limit_id。普通查询语句是一致性读，一致性读会根据 row trx_id 和 up_limit_id 的大小决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。

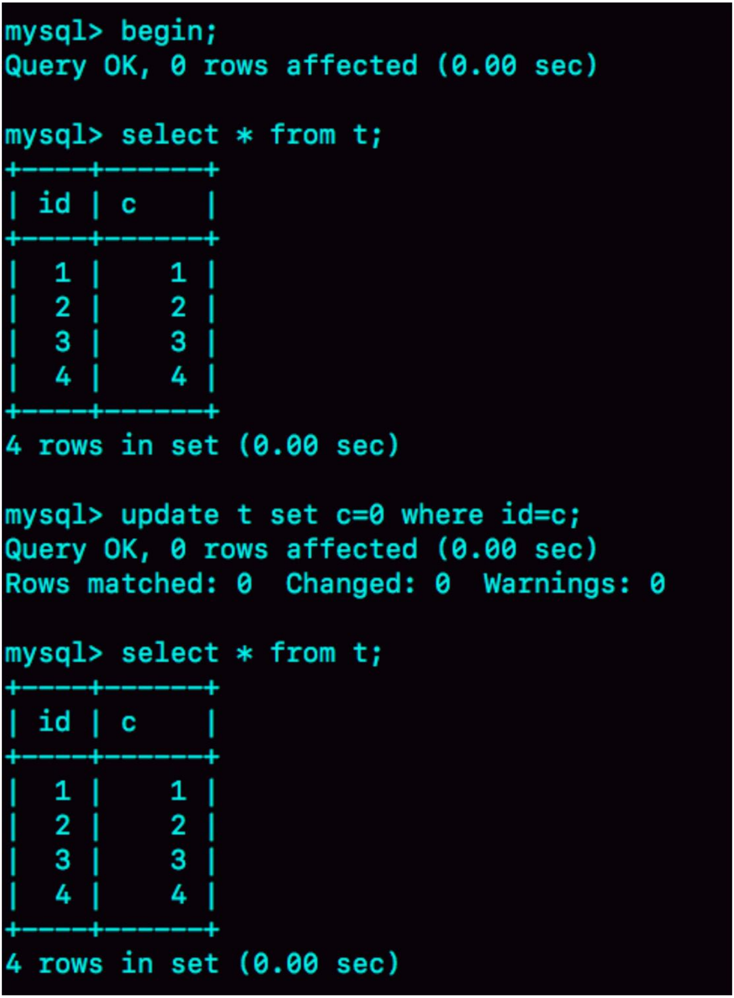
你也可以想一下，为什么表结构不支持“可重复读”？这是因为表结构没有对应的行数据，也没有 row trx_id，因此只能遵循当前读的逻辑。

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，如下图所示。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),
```



```
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？（下期解答完这个问题，我会给出数据版本可见性的完整判断规则）

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。
> @Knight^{2018} 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。

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

©版权归极客邦科技所有，未经许可不得转载

精选留言

约书亚 14

早。

思考题，RR下，用另外一个事物在update执行之前，先把所有c值修改，应该就可以。比如update t set c = id + 1。

这个实际场景还挺常见——所谓的“乐观锁”。时常我们会基于version字段对row进行cas式的更新，类似update ...set ... where id = xxx and version = xxx。如果version被其他事务抢先更新，则在自己事务中更新失败，trx_id没有变成自身事务的id，同一个事务中再次select还是旧值，就会出现“明明值没变可就是更新不了”的“异象”（anomaly）。解决方案就是每次cas

更新不管成功失败，结束当前事务。如果失败则重新起一个事务进行查询更新。
记得某期给老师留言提到了，似乎只有MySQL是在一致性视图下采用这种宽松的update机制。也许是考虑易用性吧。其他数据库大多在内部实现cas，只是失败后下一步动作有区别。

2018-11-30

作者回复

早

赞

置顶了

明天课后问题时间直接指针引用了哈😄

补充一下：上面说的“如果失败就重新起一个事务”，里面判断是否成功的标准是 affected_rows 是不是等于预期值。
比如我们这个例子里面预期值本来是4，当然实际业务中这种语句一般是匹配唯一主键，所以预期住值一般是1。

2018-11-30

夏日雨

8

老师你好，有个问题不太理解，对于文中的例子假设transaction id为98的事务在事务A执行select（Q2）之前更新了字段，那么事务A发现这个字段的row trx_id是98，比自己的up_limit_id要小，那此时事务A不就获取到了transaction id为98的事务更新后的值了吗？
换句话说对于文中"之后的更新，产生的新的数据版本的 row trx_id 都会大于 up_limit_id"这句话不太理解， up_limit_id是已经提交事务id的最大值，那也可能存在一个没有提交的id小于up_limit_id的事务对数据进行更新？还是说transaction id比up_limit_id小的事务都是保证已经提交的？

2018-11-30

作者回复

你的问题被引用最多，我回复你哈，其它同学看过来😄

好吧，今天的课后问题其实比较简单，本来是

隐藏在思考题里的彩蛋，被你问出来了哈。

Innodb 要保证这个规则：事务启动以前所有还没提交的事务，它都不可见。

但是只存一个已经提交事务的最大值是不够的。 因为存在一个问题，那些比最大值小的事务，之后也可能更新（就是你说的98这个事务）

所以事务启动的时候还要保存“现在正在执行的所有事物ID列表”，如果一个row trx_id在这列表中，也要不可见。

虽然踩破了彩蛋，还是赞你的思考哈，置顶让大家学习😄

2018-11-30

Leo 3

老师在文中说: "所以，在执行事务 B 的 Q1 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，可以用，所以 Q1 得到的 k 的值是 3。",

1. 这里不参考up_limit_id了吗?

2. 如果参考，事务B的up_limit_id是在执行update语句前重新计算的，还是在执行Q1语句前重新计算的?

2018-11-30

作者回复

1. 判断可见性两个规则：一个是up_limit_id ,另一个是“自己修改的”；这里用到第二个规则

2. 这时候事务Bup_limit_id还是99

2018-11-30

阿建

1

以下是一个错误的理解，在编写评论的过程中用前面刚学到的知识把自己的结论推翻，有一种快感，所以还是决定发出来。 哈哈~

事务A(100) | 事务B(101)

| select(1)

| update

----

update |

| select(2)

事务A B在事务启动时的up_limit_id为99
事务B update 之后表格的每一行的row_trx_id变为101
事务A 再update 之后每一行的row_trx_id变为100
事务B的select(2)时因为隔离级别是RR，所以去遍历的时候找row_trx_id<=101的版本返回，优先找到版本为100的，就会导致select(2)并没有取到自己的更新。
对于对于自己的修改也认这句话和undo-log的介绍，我觉的这种情况下会获取不到自己更新的最新的数据。不知道我理解的对不对。

不对！因为事务A的update是会被行锁锁住的，而且锁是要在事务B结束之后才释放，所以不存在在事务B的update之后还在事务中被事务A给更新，导致上面的问题。

2018-11-30

作者回复

👍🏿

我在学习过程中也是最喜欢这种“自己推翻自己结论”的快感

2018-11-30

墨萧 0

可重复读情况下，事务c的102早于事务b的101，如果事务c再get k，那不是就取得101的值了？不太明白。

2018-11-30

作者回复

咱们例子里面，事务C是直接提交的，再执行一个GET 就是另外一个事务了…

如果你说的是用begin 来启动一个多语句事务，那么事务c在更新后查询，还是看到row trx_id是102的。 【注意：如果它还没提交，101根本生成不出来，因为事务B被行锁挡着呢】

2018-11-30

某、人 5

这篇理论知识很丰富,需要先总结下
1.innodb支持RC和RR隔离级别实现是用的一致性视图(consistent read view)

2.事务在启动时会拍一个快照,这个快照是基于整个库的.

基于整个库的意思就是说一个事务内,整个库的修改对于该事务都是不可见的(对于快照读的情况)

如果在事务内select t表,另外的事务执行了DDL t表,根据发生时间,要嘛锁住要嘛报错(参考第六章)

3.事务是如何实现的MVCC呢?

(1)每个事务都有一个事务ID,叫做transaction id(严格递增)

(2)事务在启动时,找到已提交的最大事务ID记为up_limit_id。

(3)事务在更新一条语句时,比如id=1改为了id=2.会把id=1和该行之前的row trx_id写到undo log里,

并且在数据页上把id的值改为2,并且把修改这条语句的transaction id记在该行行头

(4)再定一个规矩,一个事务要查看一条数据时,必须先用该事务的up_limit_id与该行的transaction id做比对,

如果up_limit_id>=transaction id,那么可以看.

如果up_limit_id<transaction id,则只能去undo

log里去取。去undo log查找数据的时候,也需要做比对,必须up_limit_id>transaction id,才返回数据

4.什么是当前读,由于当前读都是先读后写,只能读当前的值,所以为当前读.会更新事务内的up_limit_id为该事务的transaction id

5.为什么rr能实现可重复读而rc不能,分两种情况
(1)快照读的情况下,rr不能更新事务内的up_limit_id,
而rc每次会把up_limit_id更新为快照读之前最新已提交事务的transaction id,则rc不能可重复读
(2)当前读的情况下,rr是利用record lock+gap lock来实现的,而rc没有gap,所以rc不能可重复读

2018-11-30

作者回复

👍🏿

本篇知识点全get

2018-12-01

Eric 👍 4

我不是dba，这个课程还是需要一些基础才会更有帮助，有些章节对我来说确实看起来有些吃力，但是在坚持，一遍看不懂看两遍、三遍，同时查漏补缺的去找一些资料补充盲点，还组了个一起学习的群，希望能坚持下去，收获满满

2018-11-30

作者回复

赞👍🏿

慢慢来

2018-11-30

york 👍 3

思考题为何我做出来成功修改为0了啊?

2018-11-30

作者回复

那就是没复现😄

2018-12-01

qpm 👍 1

这是典型的“丢失更新”问题。一个事务的更新操作被另外一个事务的更新操作覆盖。在RR状态下，普通select的时候是会获得旧版本数据的

，但是update的时候就检索到最新的数据。
解决方法：在读取的过程中设置一个排他锁，在 begin 事务里， select 语句中增加 for update 后缀，这样可以保证别的事务在此事务完成commit前无法操作记录。参考《MySQL技术内幕 InnoDB存储引擎》

2018-12-01

way

1

MySQL分为当前读和快照读。一般情况下，select是快照读，dml操作是当前读。RC和RR的区别就是创建read view 的时间不一样。rc在每个当前读的时候创建，rr在事物开始的时候创建。
之前遇到个问题，请老师帮忙解答下。
rr 隔离级别
数据demo
a b 都不是主键。b 上有普通索引
————————
a b
————————
1 2
10 5
—————

```
#session 1
select * from t where a = 10 ;
b=5
#session 2
select * from t where a = 10 ;
b=5
#session 1
update t set b = 100 where a = 10 ;
rows matched 1 effect 1 change 1
select * from t where a = 10 ;
#b = 100
#session 2
update t set b = 100 where a = 10 ;
--阻塞，等session 1 commit 以后
rows matched 1 effect 0 change 0
此时
#session 2
select * from t where a = 10 ;
结果是 b = 10
```

在mysql 8.0 里面测试，session2 最后的值是100 。percona server 5.7.18一下是 10 ， 所以这是低版本的bug 对吧?

2018-11-30

作者回复

是不是执行过程有误，整个过程里面并没有把b赋值成10过，怎么会查出来10?

2018-12-01

🚶______路人丙 1

老师你好第一次提问不是dba，有个疑问，事务不是先拿到锁的先执行么，为啥图3，事务C先执行?

2018-11-30

某、人 1

接上一篇更正:
rc不是没有gap.情况稍显复杂,基本上就是replace和insert有唯一索引的情况加next-key lock(还分X和S)
老师我有个问题:
假如A开启事务up_limit_id为100,B开启事务up_limit_id为110,C开启事务up_limit_id为120
A trx101
select ; B tid111
select; C tid121
update;
update commit;
commit;

select;
C最先把row tid改写为121,A随后把该行的row tid更改为101.那么B第二次去判断110>101,那不是就能查到A更改的数据了?

2018-11-30

作者回复

👍🏿

看一下置顶评论里面我的一个回复哈

2018-12-01

周巘 1

老师好 我想请问一下 如果文章中99号事务提交了 98号事务还没有提交 那么这三个事务的up_limit_id应该还是99 然后在事务A(id=100)的两次读过程中 98号事务提交并且修改了这一行 此时row trx_id=98 将会小于up_limit_id=99 那么事务A的两次读请求结果不就不一样了吗? 请老师解惑 谢谢

2018-11-30

作者回复

看一下置顶评论答复哈

2018-12-01

56 写留言 请朋友读