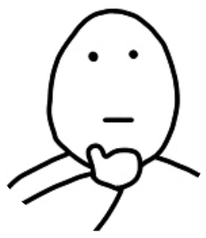


Yilun Fan

April 20, 2017 · 原创文章

MySQL加锁分析

前言

最近遇到一次MySQL死锁的问题，也算是少见的一件事情。公司的MySQL隔离级别是Read Commited，已经没有了gap lock，而且代码里的sql都再简单不过，没有显式加锁的sql语句。因此抽出时间看了一下原因。

分析具体问题之前，先整体的了解一下MySQL的加锁逻辑，之后再分析起来就游刃有余了：

MySQL的锁

为什么MySQL要加锁呢？OLTP数据库离不开事务，事务也离不开并发操作下一致性的问题。现代数据库解决事务的并发控制有两种办法，2PL和MVCC[1]。

2PL是加锁方案的代表，就是将数据操作分为加锁和解锁两个阶段，任何数据操作都会将访问对象加上锁，后续对这个对象的数据操作就会被阻塞直到锁释放（事务提交）。传统数据库大都是用2PL来实现并发控制的。

MVCC(多版本并发控制)是无锁方案的代表，通过对数据库每一次变更记录版本快照，实现读-写互不阻塞，写-写是否阻塞取决于具体实现（例如postgres的SERIALIZABLE级别下写-写互不阻塞，发生冲突抛出异常）。

对于MySQL(innoDB)来说，是通过MVCC实现读-写并发控制，又是通过2PL写-写并发控制的，因此依然保留着(悲观)锁这个概念，既然有悲观锁，自然就有可能产生死锁问题。

MySQL的事务我之前在这篇文章里做过一些粗浅的理解：**传送门**（痛心的是网上大部分资料还是显示mySql在RR隔离级别下会幻读。。）

那么MySQL会如何加锁呢[2]：

MySQL锁的模式：

- **共享/排它锁** (S锁/X锁) (Shared and Exclusive Locks)
  - S锁与X锁冲突，S锁与S锁不冲突，X锁和X锁冲突
    - 锁冲突意味着无法获取锁的事务需要等待，锁被释放后才能继续。当然也有可能等待超时或检测出死锁
  - 快照读(普通select ...)不加锁
  - select..lock in share mode / Serializable下的select 会加S锁
  - select..for update / 写操作(insert update delete) 会加X锁
  - 上述的锁都是行级别的，S锁和X锁同样可以加在表级别上，对应的语句分别是LOCK TABLE ... READ和LOCK TABLE ... WRITE
- **意向锁** (IS锁/IX锁) (Intention Locks)
  - 意向锁是**表级别**的锁，用来标识该表上面有数据被锁住（或即将被锁）
  - 一个事务在获取（任何一行/或者全表）S锁之前，一定会先在所在的表上加IS锁。同理，获取X锁之前一定会加上IX锁。
  - 意向锁提出的目的，就是要标识这个表上面有锁，这样一来，对于表级别锁的请求（LOCK TABLE ...），就可以直接判断是否有锁冲突，而不需要逐行检查锁的状态了。从更大的角度来看，意向锁就是为了实现不同粒度的锁共存，每次加锁都需要先对上面更粗粒度的数据结构加意向锁，用来表达“这个数据结构中存在被锁住的数据”。

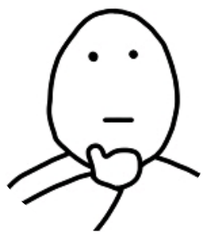
其兼容矩阵如下（+表示兼容，-表示冲突）：

\	IS	IX	S	X
IS	+	+	+	–
IX	+	+	–	–
S	+	–	+	–
X	–	–	–	–

上面提到的锁的模式，指的是如何锁住数据，各种模式之间是否兼容；下面提到的锁的类型，定义的是具体锁在哪里。二者并不冲突，比如record lock可以分成record x lock和record s lock。

MySQL锁的类型：

- Record Locks
  - 对单条索引记录上加的锁。准确的说，锁是加在索引上的而非行上。因为innodb一定会有一个聚簇索引，因此最终的行锁都会落到聚簇索引上。
  - 可以加在聚簇索引或者二级索引上。
- Gap Locks
  - gap lock是对索引间隙加的锁，可以是在一条索引记录之前，也可以在一条索引记录之后。
  - gap lock的唯一作用，就是阻止其他事务向锁住的gap里插入数据。
  - gap lock下的所有锁的模式都是兼容的，比如同一个位置的gap s lock和gap x lock是可以共存的。其作用也是完全相同的。
  - 在READ COMMITTED隔离级别下，不会使用gap lock。因此下文关于gap lock的加锁，对于RC隔离级别可以自动忽略。
- Next-Key Locks
  - Next-Key lock与record lock加锁的粒度一样，都是加在一条索引记录上的。一个next-key lock=对应的索引记录的record lock+该索引**前面的**间隙的gap lock
  - 虽然说Next-Key Lock代表着record lock+前一个间隙的gap lock，在必要的情况下，最后一条记录后面的gap也有可能作为一条单独的gap lock被锁住[3]。
  - 由于锁住的是前面的间隙，所以有些资料也会用左开右闭的区间来表示next-key lock，例如(1,3]
- Insert Intention Locks
  - Insert Intention Lock是一种特殊的间隙锁，执行insert之前会向插入的间隙加上Insert Intention Lock



Yilun Fan

- Insert Intention Lock与已有的gap lock冲突，因此gap lock锁住的间隙是不能插入数据的
- Insert Intention Lock与Insert Intention Lock之间不冲突，因此允许了同时向同一个间隙插入不同主键的数据

其兼容矩阵如下，+表示兼容，-表示冲突：

要加的锁\ 已存在的锁	record lock	gap lock	insert intention lock	next key lock
record lock	-	+	+	-
gap lock	+	+	+	+
insert intention lock	+	-	+	-
next key lock	-	+	+	-

如何查看事务的加锁情况

当存在锁冲突/等待时，比较方便的查看锁冲突的方式：

```
// innodb_locks记录了所有innodb正在等待的锁，和被等待的锁
select * from information_schema.innodb_locks;
// innodb_lock_waits记录了所有innodb锁的持有和等待关系
select * from information_schema.innodb_lock_waits'
```

select * from information_schema.innodb_locks;

lock_id	lock_trx_id	lock_mode	lock_type	lock_table	lock_index	lock_space	lock_page	lock_rec	lock_data
4580:37:3:4	4580	X	RECORD	`svs`.`new table`	PRIMARY	37	3	4	3
4579:37:3:4	4579	X	RECORD	`svs`.`new table`	PRIMARY	37	3	4	3

select * from information_schema.innodb_lock_waits;

requesting_trx_id	requested_lock_id	blocking_trx_id	blocking_lock_id
4580	4580:37:3:4	4579	4579:37:3:4

结果如上图，可以看到当前事务id 4579持有着'new_table'表的聚簇索引=3的X锁。事务id 4580正在等待'new_table'表的聚簇索引=3的X锁。

但是上述方式只能看到存在锁冲突的记录，不能看到每个事务实际锁住的记录和范围。因此更通用的办法是，直接打开innodb的锁监控，在控制台查看详细锁状态：

```
mysql> set global innodb_status_output=ON; // 可选。将监控输出到log_error输出中，15秒刷新一次
mysql> set global innodb_status_output_locks=ON; // 输出的内容包含锁的详细信息
```

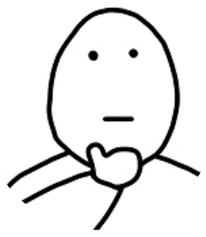
通过show engine innodb status;语句，可以输出每个事务当前持有的锁结果，常见的结果类型解释如下。死锁日志也会记录如下的锁记录，因此可以用同样的方式来读MySQL的死锁日志。

```
// 表示事务4641对表`sys`.`new_table`持有了IX锁
TABLE LOCK table `sys`.`new_table` trx id 4641 lock mode IX

// space id=38，space id可以唯一确定一张表，表示了锁所在的表
// page no 3，表示锁所在的页号
// index PRIMARY 表示锁位于名为PRIMARY的索引上
// lock_mode X locks rec but not gap 表示x record lock
// 下方的数据表示了被锁定的索引数据，最上面一行代表索引列的十六进制值，在这里表示的就是id=3的数据
RECORD LOCKS space id 38 page no 3 n bits 80 index PRIMARY of table `sys`.`new_table` trx id 4641 lock_mode X loc
ks rec but not gap
Record lock, heap no 4 PHYSICAL RECORD: n_fields 8; compact format; info bits 0
 0: len 4; hex 00000003; asc     ;;
 1: len 6; hex 0000000011e9; asc       ;;
 2: len 7; hex a70000011b0128; asc       (;;
 3: len 4; hex 8000012c; asc    ,;;
 4: len 1; hex 63; asc c;;
 5: len 4; hex 80000006; asc     ;;
 6: len 3; hex 636363; asc ccc;;
 7: len 2; hex 3333; asc 33;;

// lock_mode X表示的是next-key lock，即当前记录的record lock+前一个间隙的gap lock
// 这个锁在名为idx1的索引上，对应的索引列的值为100（hex 64对应十进制），对应聚簇索引的值为1
RECORD LOCKS space id 38 page no 5 n bits 80 index idx1 of table `sys`.`new_table` trx id 4643 lock_mode X
Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 00000064; asc    d;;
 1: len 4; hex 00000001; asc     ;;

// lock_mode X locks gap before rec表示的是对应索引记录前一个间隙的gap lock
RECORD LOCKS space id 38 page no 5 n bits 80 index idx1 of table `sys`.`new_table` trx id 4643 lock_mode X locks
gap before rec
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 4; hex 800000c8; asc     ;;
 1: len 4; hex 00000002; asc     ;;
```



Yilun Fan


```
0
3
1
```

不同语句的加锁情况

以下实验数据基于MySQL 5.7。
假设已知一张表my_table，id列为主键。

id	name	num
1	aaa	100
5	bbb	200
8	bbb	300
10	ccc	400

对该表进行读写操作，可能产生的加锁情况如下（仅考虑隔离级别为RR和RC）：

1. 查询命中聚簇索引（主键索引）

1.1 如果是精确查询，那么会在命中的索引上加record lock。
例如：

```
// 在id=1的聚簇索引上加X锁
update my_table set name='a' where id=1;

// 在id=1的聚簇索引上加S锁
select * from my_table where id=1 lock in share mode;
```

1.2 如果是范围查询，那么

- 1.2.1 在RC隔离级别下，会在所有命中的行的聚簇索引上加record locks（只锁行）

```
// 在id=8和10的聚簇索引上加X锁
update my_table set name='a' where id>7;

// 在id=1的聚簇索引上加X锁
update my_table set name='a' where id<=1;
```

- 1.2.2 在RR隔离级别下，会在所有命中的行的聚簇索引上加next-key locks（锁住行和间隙）。最后命中的索引的后一条记录，也会被加上next-key lock。

```
// 在id=8、10(、+∞)的聚簇索引上加X锁
// 在(5,8)(8,10)(10,+∞)加gap lock
update my_table set name='a' where id>7;

// 在id=1、5的聚簇索引上加X锁
// 在(-∞,1)(1,5)加gap lock
update my_table set name='a' where id<=1;
```

1.3 如果查询结果为空，那么

- 1.2.1 在RC隔离级别下，什么也不会锁
- 1.2.2 在RR隔离级别下，会锁住查询目标所在的间隙。

```
// 在(1,5)加gap lock
update my_table set name='a' where id=2;
```

2. 查询命中唯一索引

假设上述表中，num列加了唯一索引
2.1 如果是精确查询，那么会在命中的唯一索引，和对应的聚簇索引上加record lock。

```
// 在num=100的唯一索引上加X锁
// 并在id=1的聚簇索引上加X锁
update my_table set name='a' where num=100;
```

2.2 如果是范围查询，那么

- 2.2.1 在RC隔离级别下，会在所有命中的唯一索引和聚簇索引上加record lock。同2.1
- 2.2.2 在RR隔离级别下，会在所有命中的行的唯一索引上加next-key locks。最后命中的索引的后一条记录，也会被加上next-key lock。

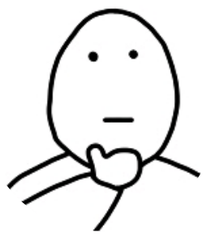
```
// 在num=100和num=200的唯一索引上加X锁
// 并在id=1和id=5的聚簇索引上加X锁
// 并在唯一索引的间隙(-∞,100)(100,200)加gap lock
update my_table set name='a' where num<150;
```

2.3 如果查询结果为空，同1.3。唯一差别在于，此时加的gap lock是位于唯一索引上的。

3. 查询命中二级索引（非唯一索引）

假设上述表中，name列加了普通二级索引，num列没有索引
3.1 如果是精确查询，那么

- 3.1.1 在RC隔离级别下，同2.1，对命中的二级索引和聚簇索引加record lock



Yilun Fan


```
// 在name='bbb'的两条索引记录上加X锁
// 并在id=5和id=8的聚簇索引上加X锁
update my_table set num=10 where name='bbb';
```

- 3.1.2 在RR隔离级别下，会在命中的二级索引上加next-key lock，最后命中的索引的后面的间隙会加上gap lock。对应的聚簇索引上加record lock。

```
// 在name='bbb'的两条索引记录上加X锁
// 并在id=5和id=8的聚簇索引上加X锁
// 并在二级索引的间隙('aaa','bbb')('bbb','bbb')('bbb','ccc')加gap lock
update my_table set num=10 where name='bbb';
```

3.2 范围查询、模糊查询的情况比较复杂，此处不详述。可以用上述方法自己实验。

4. 查询没有命中索引

假设上述表中，name列加了普通二级索引，num列没有索引
4.1 如果查询条件没有命中索引

- 4.1.1 在RC隔离级别下，对命中的数据的聚簇索引加X锁。根据MySQL官方手册[4]，对于update和delete操作，RC只会锁住真正执行了写操作的记录，这是因为尽管innodb会锁住所有记录，MySQL Server层会进行过滤并把不符合条件的锁当即释放掉[5]。同时对于UPDATE语句，如果出现了锁冲突（要加锁的记录上已经有锁），innodb不会立即锁等待，而是执行semi-consistent read：返回改数据上一次提交的快照版本，供MySQL Server层判断是否命中，如果命中了才会交给innodb锁等待。因此加锁情况可以这样来认为：

```
// 在id=5的聚簇索引上加X锁
update my_table set num=1 where num=200;

// 先在id=1,5,8,10（全表所有记录）的聚簇索引上加X锁
// 然后马上释放id=1,8,10的锁，只保留id=5的锁
delete from my_table where num=200;
```

- 4.1.2 在RR隔离级别下，事情就很糟糕了，对全表的所有聚簇索引数据加next-key lock

```
// 在id=1,5,8,10（全表所有记录）的聚簇索引上加X锁
// 并在聚簇索引的所有间隙(-∞,1)(1,5)(5,8)(8,10)(10,+∞)加gap lock
update my_table set num=100 where num=200;

// 尽管name列有索引，但是like '%%'查询不使用索引，因此此时也是锁住所有聚簇索引，情况和上面一模一样
update my_table set num=100 where name like '%b%';
```

5. 对索引键值有修改

假设上述表中，num列加了二级索引

如果一条update语句，对索引键值有修改，那么修改前后的数据如何加锁呢。这点要结合数据多版本的可见性来考虑：无论是聚簇索引，还是二级索引，只要其键值更新，就会产生新版本。将老版本数据deleted bti设置为1；同时插入新版本[6]。因此可以认为，一次索引键值的修改实际上操作了两条索引数据：原索引和修改后的新索引。

从innodb的事务的角度来看，如果一个事务操作（写）了一条数据，那么这条数据一定要加锁。因此可以认为，**如果修改了索引键值，那么修改前和修改后的索引都会加锁**。另外，由于修改的数据并没有被作为查询条件，那么也不会有“不可重复读”和“幻读”的问题，因此无需加gap lock，索引修改只会加X record lock。

示例（RC和RR级别效果一样）：

```
// 在id=1的聚簇索引上加X锁
// 并在name='aaa'（name列索引原键值）和name='eee'（新键值）的索引上加锁
update my_table set name='eee' where id=1;
```

6. 插入数据

假设上述表中，num列加了二级索引
insert加锁过程：

1. 唯一索引冲突检查：表中一定有至少一个唯一索引，那么首先会做唯一索引的冲突检查。innodb检查唯一索引冲突的方式是，对目标的索引项加S锁（因为不能依赖快照读，需要一个彻底的当前读），读到数据则唯一索引冲突，返回异常，否则检查通过。
2. 对插入的间隙加上插入意向锁(Insert Intention Lock)
3. 对插入记录的所有索引项加X锁

示例：

```
// 先对id=15加S锁
// 再对间隙id(10,+∞)和name('ccc',+∞)加Insert Intention Lock
// 然后在id=15的聚簇索引上加X锁（S锁升级为X锁）
// 并在name='fff'的索引上加X锁
insert into my_table (`id`, `name`, `num`) values ('15', 'fff', '800');
```

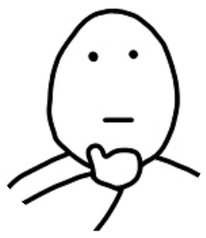
还有一个有趣的问题，如果插入的二级索引键值已经存在，那么这个插入意向锁会加在哪个间隙中呢？

顾名思义，插入意向锁锁定的间隙一定是将要插入的索引的位置，如果二级索引键值相同，默认会按照聚簇索引的大小来排序（二级索引在存储上其实就是{索引值,主键值}）。例如：

```
// 插入意向锁加在间隙 ({'aaa',1},{'bbb',5}) 上
insert into my_table (`id`, `name`, `num`) values ('4', 'bbb', '800');

// 插入意向锁加在间隙 ({'bbb',5},{'bbb',8}) 上
insert into my_table (`id`, `name`, `num`) values ('6', 'bbb', '800');

// 插入意向锁加在间隙 ({'bbb',8},{'ccc',10}) 上
insert into my_table (`id`, `name`, `num`) values ('11', 'bbb', '800');
```



Yilun Fan

隐式锁

为了降低锁的开销，innodb采用了延迟加锁机制，即隐式锁(implicit lock)[7]。

从数据存储结构上看，每张表的数据都是挂在聚簇索引的B+树下面的叶子节点上（每个节点代表一个page，每个page存放着多行数据）。每行存储的信息项中都会存有一隐藏列事务id。当有事务对这条记录进行修改时，需要先判断该行记录是否有隐式锁（原记录的事务id是否是活动的事务），如果有则为其真正创建锁并等待，否则直接更新数据并写入自己的事务id。

二级索引虽然存储上没有记录事务id，但同样可以存在隐式锁，只不过判断逻辑复杂一些，需要依赖对应的聚簇索引做计算。

当然，隐式锁只是一个实现细节，显示还是隐式加锁并不影响上文对加锁的判断。

另外，聚簇索引每行记录的事务id，还有一个重要作用就是实现MVCC快照读：由于事务id是全局递增的，那么进行快照读的时候，如果数据的事务id小于当前事务id并且不在活跃事务列表内（尚未提交），则直接返回当前行数据。否则需要根据roll pointer（和事务id一样，也在每行的隐藏列中）去查找undo日志。

一个RC隔离级别下的死锁

其实可以看到，RC隔离级别下的加锁已经很少了，用官方文档的话说”greatly reduces the probability of deadlocks”。因此尽管MySQL的默认隔离级别是RR，但是互联网应用更倾向与使用RC来避免死锁+提高并发能力。例如阿里电商的MySQL默认级别就是RC。

尴尬的是，但是我也的的确确碰到了RC的死锁。还是以这个表来举例，假设id为主键，num列无索引。

id	name	num
1	aaa	100
5	bbb	200
8	bbb	300

按以下顺序执行事务：

trx1	trx2
insert into my_table (id, name, num) values (‘16’, ‘rrr’, ‘888’);	-

- \| insert into my_table (id, name, num) values (‘17’, ‘ttt’, ‘999’);
  delete from sys.my_table where num=300; // waiting \| -
- \| delete from sys.my_table where num=400; // deadlock

对照上文的加锁逻辑，insert会对聚簇索引加X锁，因此trx1和trx2首先会分别持有id=16和id=17的X锁。

接下来坑爹的事情来了，对于无索引字段，delete操作不会执行semi-consistent read，而是先直接锁住所有数据的聚簇索引（尽管后面会马上释放，但也需要先获取锁）。这样一来，事务1的delete需要锁住所有记录，等待事务2持有的id=17的X锁，而事务2的delete需要等待事务1的id=16的X锁。死锁就产生了。

在这个例子中，如果insert和delete的顺序都颠倒一下，或者delete都变为update，死锁都不会发生。

小结

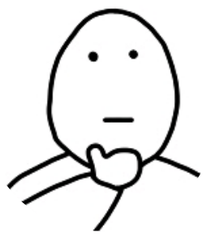
- 索引记录的间隙上用来避免幻读。
- Select（Serializable隔离级别除外）不会加锁，而是执行快照读。
- 写操作都会加锁，具体加锁方式取决于隔离级别、索引命中情况以及修改的索引情况。
- 为了减少锁的范围，避免死锁的发生，应该尽量让查询条件命中索引，而且命中的越精确加锁越少。同时如果能接受RC级别对一致性的破坏，可以将隔离级别调整成RC。

参考资料

[1] 萧美阳, 叶晓俊. 并发控制实现方法的比较研究[J]. 计算机应用研究, 2006, 23(6):19-22.
[2] MySQL 5.7 Reference Manual :: 15.5.1 InnoDB Locking
[3] MySQL 5.7 Reference Manual :: 15.5.4 Phantom Rows
[4] MySQL 5.7 Reference Manual :: 15.5.2.1 Transaction Isolation Levels
[5] MySQL **加锁处理分析**
[6] InnoDB**多版本(MVCC)实现简要分析**
[7] Introduction to Transaction Locks in InnoDB Storage Engine

PPT参考

同主题分享做的PPT，贴出来供参考：



Yilun Fan

← prev • next →