目录

0. 背景

在业务低峰通过pt-osc在线做DDL期间出现死锁，导致业务的SQL被回滚了，对应用不友好。

本案例死锁发生的场景：pt-osc拷贝最后一个chunk-size并且期间其它事务有对原表做insert操作，才会出现本案例的死锁。

1. 先简单介绍一下pt-osc的工作原理

1. 创建一个跟原表表结构一样的新表；
2. 如果是添加/删除字段，会修改新表的表结构；
3. 在原表中创建insert、update、delete这3个类型的触发器，用于增量数据的迁移；原SQL和触发器触发的SQL在同一个事务里。
4. 会以一定块大小(chunk-size)从原表拷贝数据到新表中；
5. 数据拷贝完成之后，rename表，整个过程为原子操作：原表重命名为old表，新表重命名为原表；
6. 删除old表(默认)，删除3个触发器。整个过程只在rename表的时间会锁一下表，其他时候不锁表；
7. pt-osc操作完成。

2. 环境

MySQL版本为 5.7.26；

事务隔离级别为RC读已提交；

参数innodb_autoinc_lock_mode=1；

先介绍一下数据表情况，因为涉及到公司内部真实的数据，所以表结构和死锁日志都做了脱敏操作，但不会影响具体的分析。

3. 表的DDL

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `c1` int(11) NOT NULL DEFAULT '0' COMMENT 'c1',
  `c2` int(11) NOT NULL DEFAULT '0' COMMENT 'c2'
  `c3` int(11) NOT NULL DEFAULT '0' COMMENT 'c3'
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 COMMENT='';
```

4. 死锁日志分析

4.1 生产环境中死锁现场的日志

```
2020-04-26T06:24:05.340343+08:00 733947 [Note] InnoDB: Transactions deadlock
detected, dumping detailed information.
2020-04-26T06:24:05.341246+08:00 733947 [Note] InnoDB:
*** (1) TRANSACTION:

TRANSACTION 918773485, ACTIVE 0 sec setting auto-inc lock
mysql tables in use 2, locked 2
LOCK WAIT 4 lock struct(s), heap size 1136, 1 row lock(s), undo log entries 2
MySQL thread id 668554, OS thread handle 139777592952576, query id 2675769996
192.168.1.1 test_user update
REPLACE INTO `test_db`.`_t_new` (`id`, `c1`, `c2`, `c3`) VALUES (NEW.`id`,
NEW.`c1`, NEW.`c2`, NEW.`c3`)
2020-04-26T06:24:05.341319+08:00 733947 [Note] InnoDB: *** (1) WAITING FOR THIS
LOCK TO BE GRANTED:

TABLE LOCK table `test_db`.`_t_new` trx id 918773485 lock mode AUTO-INC waiting

2020-04-26T06:24:05.341351+08:00 733947 [Note] InnoDB: *** (2) TRANSACTION:

TRANSACTION 918773482, ACTIVE 1 sec fetching rows
mysql tables in use 2, locked 2
120 lock struct(s), heap size 24784, 8894 row lock(s), undo log entries 8099
MySQL thread id 733947, OS thread handle 139777597007616, query id 2675769985
localhost root Sending data
INSERT LOW_PRIORITY IGNORE INTO `test_db`.`_t_new` (`id`, `c1`, `c2`, `c3`)
SELECT `id`, `c1`, `c2`, `c3` FROM
`test_db`.`t` FORCE INDEX(`PRIMARY`) WHERE ((`id` >= '95439963')) AND ((`id` <=
'95448404')) LOCK IN SHARE MODE
2020-04-26T06:24:05.341398+08:00 733947 [Note] InnoDB: *** (2) HOLDS THE
LOCK(S):

TABLE LOCK table `test_db`.`_t_new` trx id 918773482 lock mode AUTO-INC
2020-04-26T06:24:05.341415+08:00 733947 [Note] InnoDB: *** (2) WAITING FOR THIS
LOCK TO BE GRANTED:

RECORD LOCKS space id 974 page no 145414 n bits 80 index PRIMARY of table
`test_db`.`t` trx id 918773482 lock mode S locks rec but not gap waiting
Record lock, heap no 9 PHYSICAL RECORD: n_fields 27; compact format; info bits 0
0: len 4; hex 85b06d55; asc    mU;; --'5b06d55'从16进制转换为10进制,得到的值为
95448405
```

```
1: len 4; hex 80002712; asc    ;;
2: len 4; hex 800c24d7; asc   $ ;;
3: len 4; hex 80000003; asc     ;;

2020-04-26T06:24:05.342491+08:00 733947 [Note] InnoDB: *** WE ROLL BACK
TRANSACTION (1)
```

4.2 死锁的分析

TRANSACTION 918773482 的信息：

```
当前的SQL语句:
    INSERT LOW_PRIORITY IGNORE INTO `test_db`.`_t_new` (`id`, `c1`, `c2`, `c3`)
SELECT `id`, `c1`, `c2`, `c3` FROM
    `test_db`.`t` FORCE INDEX(`PRIMARY`) WHERE ((`id` >= '95439963')) AND ((`id`
<= '95448404')) LOCK IN SHARE MODE;

持有的锁信息:
    TABLE LOCK table `test_db`.`_t_new` ... lock mode AUTO-INC --表示持有表_t_new
上的自增长锁;

在等待的锁信息:
     index PRIMARY of table `test_db`.`t` --表示在等的是表t的主键索引上面的锁;
     lock_mode S locks rec but not gap waiting --表示需要加一个共享锁(读锁)，当前的状
态是等待中;
     0: len 4; hex 85b06d55; asc    mU;;   --主键字段, 5b06d55的16进制为转换为10进制
得到值为: 95448405;

通过分析得知:
    TRANSACTION 918773482持有的锁: 表_t_new的自增长锁;
    TRANSACTION 918773482在等待TRANSACTION 918773485的锁: 表t的主键索引primary:
record lock: id=95448405。
```

TRANSACTION 918773485 的信息：

```
当前的SQL语句:
    REPLACE INTO `test_db`.`_t_new` (`id`, `c1`, `c2`, `c3`) VALUES (NEW.`id`,
NEW.`c1`, NEW.`c2`, NEW.`c3`);

持有的锁信息:
    根据TRANSACTION 918773482在等待TRANSACTION 918773485的锁为 primary: record
lock: id=95448405 的行锁，所以推导出TRANSACTION 918773485持有表t主键索引 id=95448405
的行锁;

在等待的锁信息:
    TABLE LOCK table `test_db`.`_t_new` ... lock mode AUTO-INC waiting --表示在等
表_t_new上的自增长锁;

通过分析得知:
    TRANSACTION 918773485持有的锁：持有表t主键索引primary id=95448405 的行锁;
    TRANSACTION 918773485在等待TRANSACTION 918773482的锁: 表_t_new上的自增长锁。
```

4.3 根据分析死锁日志和pt-osc原理得到事务的执行次序
根据pt-osc的原理得知：原SQL和触发器触发的SQL在同一个事务里， `TRANSACTION 918773485` 的T2时刻的语句就是原SQL，而T3时刻的语句就是触发器触发的SQL。

时间点	TRANSACTION 918773482	TRANSACTION 918773485
	begin;	begin;
	INSERT LOW_PRIORITY IGNORE INTO `_t_new` (`id` , `c1` , `c2` , `c3`) SELECT `id` , `c1` , `c2` , `c3` FROM `test_db` . `t` FORCE INDEX(`PRIMARY`) WHERE ((`id` >= '95439963')) AND ((`id` <= '95448404')) LOCK IN SHARE MODE;	
T1	持有的锁: 表_t_new: AUTO-INC	
T2		INSERT INTO `t` (`c1` , `c2` , `c3`) VALUES (0, 0, 0); --持有表t主键索引 id=95448405 的行锁(排他X锁)
T3		REPLACE INTO `_t_new` (`id` , `c1` , `c2` , `c3`) VALUES (NEW. `id` , NEW. `c1` , NEW. `c2` , NEW. `c3`); --在等待的锁: 表_t_new: AUTO-INC
T4	在等待的锁: 表t: primary: record lock: id=95448405	

T3被T1阻塞，T4被T2阻塞，因此锁资源请求形成了环路，进而触发死锁检测，MySQL会把执行代价最小的事务回滚掉，让其它事务得以继续进行;
这里是把 `TRANSACTION 918773485` 也就是业务的SQL回滚掉，对应用不友好。

4.4 引发的思考

`TRANSACTION 918773485` 在T2时刻持有主键 id=95448405 的行锁；

`TRANSACTION 918773482` 的 `WHERE ((id >= '95439963')) AND ((id <= '95448404')) LOCK IN SHARE MODE` ; 为什么在RC隔离级别下需要申请持有主键 id=95448405 的行锁？

因为 id <= 95448404 是范围等值查询并且id=95448404是当前主键索引的最大值 ， 锁的过程实际上是id<=95448404的下一条记录也就是这个索引页的最大记录supremum（如果这个在RC隔离级别下没有被锁，则会立即释放），需要访问到 id=95448405 才会停止下来，所以需要申请 持有 id=95448405 的行锁 ，因此被 T2时刻 的SQL语句阻塞。

`TRANSACTION 918773482` 的事务语句是pt-osc拷贝的最后一个chunk-size，并且期间其它事务有对原表做insert操作， 所以才会发生死锁。

5. 死锁的复现

5.1 表结构和数据初始化

```
DROP TABLE IF EXISTS `t`;
CREATE TABLE `t` (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `t_new`;
CREATE TABLE `t_new` (
  `id` bigint(11) NOT NULL AUTO_INCREMENT,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4;

#往表t写入50W行记录
DROP PROCEDURE IF EXISTS `test_data`;
DELIMITER ;;
CREATE DEFINER=`root`@`localhost` PROCEDURE `test_data`()
begin
  declare i int;
  set i=1;
  start transaction;
  while(i<=500000) do
    INSERT INTO t (c,d) values (i,i);
    set i=i+1;
  end while;
  commit;
end
;;
DELIMITER ;

call test_data();

mysql> select count(*) from t;
+----------+
| count(*) |
+----------+
|   500000 |
+----------+
1 row in set (0.36 sec)
```

注意： 这里用50W记录做实验的原因：因为是手工模拟pt-osc产生的死锁，如果数据量太小，那么从原表批量迁移数据到新表的SQL语句执行速度过快，可能会形成不了事务之间锁等待的现象， 导致手工模拟不出死锁的现象。

5.2 事务的执行次序

事务1(重建表批量迁移语句)	事务2(业务的插入语句)
begin;	begin;
INSERT LOW_PRIORITY IGNORE INTO `t_new` (`id`, `c`, `d`) SELECT `id`, `c`, `d` from t WHERE ((`id` >= '1')) AND ((`id` <= '500000')) LOCK IN SHARE MODE;	
	INSERT INTO `t` (`c`, `d`) VALUES ('500001', '500001');
	replace INTO `t_new` (`id`, `c`, `d`) VALUES (500001, '500001', '500001'); (Blocked)
Query OK, 500000 rows affected (5.81 sec)	
	ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

5.3 死锁日志

```
2020-05-21T16:54:27.687689+08:00 3 [Note] InnoDB: Transactions deadlock
detected, dumping detailed information.
2020-05-21T16:54:27.687728+08:00 3 [Note] InnoDB:
*** (1) TRANSACTION:

TRANSACTION 166084117, ACTIVE 4 sec setting auto-inc lock
mysql tables in use 1, locked 1
LOCK WAIT 4 lock struct(s), heap size 1136, 1 row lock(s), undo log entries 2
MySQL thread id 4, OS thread handle 140194130835200, query id 42 localhost root
update
replace INTO `t_new` (`id`, `c`, `d`) VALUES (500001, '500001', '500001')
2020-05-21T16:54:27.687791+08:00 3 [Note] InnoDB: *** (1) WAITING FOR THIS LOCK
TO BE GRANTED:

TABLE LOCK table `sbtest`.`t_new` trx id 166084117 lock mode AUTO-INC waiting
2020-05-21T16:54:27.687816+08:00 3 [Note] InnoDB: *** (2) TRANSACTION:

TRANSACTION 166084112, ACTIVE 6 sec fetching rows
mysql tables in use 2, locked 2
1173 lock struct(s), heap size 172240, 500001 row lock(s), undo log entries
500000
MySQL thread id 3, OS thread handle 140194131363584, query id 40 localhost root
Sending data
INSERT LOW_PRIORITY IGNORE INTO `t_new` (`id`, `c`, `d`) SELECT `id`, `c`, `d`
from t WHERE ((`id` >= '1')) AND ((`id` <= '500000')) LOCK IN SHARE MODE
2020-05-21T16:54:27.687845+08:00 3 [Note] InnoDB: *** (2) HOLDS THE LOCK(S):

TABLE LOCK table `sbtest`.`t_new` trx id 166084112 lock mode AUTO-INC
2020-05-21T16:54:27.687861+08:00 3 [Note] InnoDB: *** (2) WAITING FOR THIS LOCK
TO BE GRANTED:
```

```
RECORD LOCKS space id 296 page no 1840 n bits 384 index PRIMARY of table
`sbtest`.`t` trx id 166084112 lock mode S locks rec but not gap waiting

Record lock, heap no 312 PHYSICAL RECORD: n_fields 5; compact format; info bits
0
 0: len 8; hex 800000000007a121; asc        !;;  --'7a121'从16进制转换为10进制,得到
的值为 500001
 1: len 6; hex 000009e63e15; asc     > ;;
 2: len 7; hex b3000002960110; asc        ;;
 3: len 4; hex 8007a121; asc    !;;
 4: len 4; hex 8007a121; asc    !;;

2020-05-21T16:54:27.688082+08:00 3 [Note] InnoDB: *** WE ROLL BACK TRANSACTION
(1)
```

可以看到，死锁重现了，跟通过pt-osc在线重建表导致死锁的信息一致。

5.4 死锁关系

时间点	TRANSACTION 166084112	TRANSACTION 166084117
	begin;	begin;
	INSERT LOW_PRIORITY IGNORE INTO `t_new` (`id`, `c`, `d`) SELECT `id`, `c`, `d` from t WHERE ((`id` >= '1')) AND ((`id` <= '500000')) LOCK IN SHARE MODE;	
T1	持有的锁: 表t_new: AUTO-INC	
T2		INSERT INTO `t` (`c`, `d`) VALUES ('500001', '500001'); --持有的锁: 表t: primary: record lock: id=500001
T3		replace INTO `t_new` (`id`, `c`, `d`) VALUES (500001, '500001', '500001'); --在等待的锁: 表t_new: AUTO-INC
T4	在等待的锁: 表t: primary: record lock: id=500001	

T3被T1阻塞，T4被T2阻塞，因此锁资源请求形成了环路，进而触发死锁检测，MySQL会把执行代价最小的事务回滚掉，让其它事务得以继续进行。

6. 优化本案例死锁的几种方案

1. 设置pt-osc的chunk-size为更小的值，可以减少死锁的发生，但是不可能避免死锁的发生。
2. 如果参数innodb_autoinc_lock_mode的值为2，大大降低死锁发生的概率，原因如下：
   造成本案例死锁的原因之一就是在参数innodb_autoinc_lock_mode=1的环境下，持有的自增锁直到SQL语句结束后才释放；
   如果参数innodb_autoinc_lock_mode=2，自增锁在申请后就释放，不需要等语句结束，大大缩短

了持有自增锁的时间，从而降低了死锁发生的概率。
在MySQL 5.7版本，该参数的默认值为1；在MySQL 8.0版本，该参数的默认值为2。
如果设置参数值为2，binlog格式一定要为row，不然可能会出现主从数据不一致的情况。

3. 数据库版本为MySQL 8.0.18或者以上， 事务隔离为RR可重复读则不会出现本案例的死锁，原因如下：
   8.0.18或者以上的版本中，对加锁规则有一个优化：在RR可重复读级别下，唯一索引上的范围查询，不再需要访问到不满足条件的第一个值为止（即不再需要对不必要的数据上锁）。在叶老师的这篇文章中有说明：https://mp.weixin.qq.com/s/xDKKulvVgFNiKp5kt2NIgA InnoDB这个将近20年的"bug"修复了；

   本案例的 `TRANSACTION 166084112` 的 `SELECT `id`, `c`, `d` from t WHERE ((`id` >= '1')) AND ((`id` <= '500000')) LOCK IN SHARE MODE;` 语句访问到 id=500000 的记录会停止下来，不再需要访问到 id=500001的这一行记录(即不再需要对id=500001的记录上锁)，因此不会有锁等待，那么就不会产生本案例的死锁。
4. 使用gh-ost工具来做表的DLL，gh-ost无需创建触发器，自然不会产生本案例的死锁。

7. 小结

1. 通过分析，验证了在文章开头提到的 “pt-osc拷贝最后一个chunk-size并且期间其它事务有对原表做insert操作，才会出现本案例的死锁” 的这一结论。
2. 发生死锁不可怕， 一般可以结合死锁现场的日志、加锁规则和业务场景等做相关的分析并进行优化。

由于水平有限，文章可能会存在错误，还望大家能够指出问题。
全文完。