# 概述

死锁是并发系统中常见的问题，同样也会出现在数据库系统的并发读写请求场景中。当两个及以上的事务，双方都在等待对方释放已经持有的锁或因为加锁顺序不一致造成循环等待锁资源，就会出现"死锁"。

举例来说A 事务持有X1锁，申请X2锁，B事务持有X2锁，申请X1锁。A和B事务持有锁并且申请对方持有的锁进入循环等待，就造成死锁。

从死锁的定义来看，MySQL出现死锁的几个要素:

1、两个或者两个以上事务。

2、每个事务都已经持有锁并且申请新的锁。

3、锁资源同时只能被同一个事务持有或者不兼容。

4、事务之间因为持有锁和申请锁导致彼此循环等待。

# 事务加锁类型

例子: update tab set x=1 where id= 1 ;

1. 索引列是主键，RC隔离级别,对记录记录加X锁
2. 索引列是二级唯一索引，RC隔离级别。若id列是unique列，其上有unique索引。那么SQL需要加两个X锁，一个对应于id unique索引上的id = 10的记录，另一把锁对应于聚簇索引上的[name='d',id=10]的记录。
3. 索引列是二级非唯一索引，RC隔离级别 若id列上有非唯一索引，那么对应的所有满足SQL查询条件的记录，都会被加锁。同时，这些记录在主键索引上的记录，也会被加锁。
4. 索引列上没有索引，RC隔离级别

若id列上没有索引，SQL会走聚簇索引的全扫描进行过滤，由于过滤是由MySQL Server层面进行的。因此每条记录，无论是否满足条件，都会被加上X锁。但是，为了效率考量，MySQL做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了2PL的约束。

索引列是主键，RR隔离级别 对记录记录加X锁

索引列是二级唯一索引，RR隔离级别 对表加上两个X锁，唯一索引满足条件的记录上一个，对应的聚簇索引上的记录一个。

索引列是二级非唯一索引，RR隔离级别

结论:Repeatable Read隔离级别下，id列上有一个非唯一索引，对应SQL:delete from t1 where id = 10;

首先，通过id索引定位到第一条满足查询条件的记录，加记录上的X锁，加GAP上的GAP锁，然后加主键聚簇索引上的记录X锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11,f]，此时，不需要加记录X锁，但是仍旧需要加GAP锁，最后返回结束。

索引列上没有索引，RR隔离级别则锁全表

这里需要重点说明insert 和delete的加锁方式，因为目前遇到的大部分案例或者部分难以分析的案例都是和delete，insert 操作有关。

## insert加锁方式

insert的流程(有唯一索引的情况): 比如insert N

1. 找到大于N的第一条记录M，以及前一条记录P
2. 如果M上面没有gap/next-key lock，进入第三步骤，否则等待(对其next-rec加insert intension lock，由于有gap锁，所以等待)
3. 检查P：判断P是否等于N:

如果不等: 则完成插入（结束）

如果相等: 再判断P是否有锁，

a 如果没有锁:报1062错误(duplicate key)，说明该记录已经存在，报重复值错误；

b 加S-lock,说明该记录被标记为删除, 事务已经提交，还没来得及purge；

c 如果有锁: 则加S-lock，说明该记录被标记为删除，事务还未提交。

## delete加锁方式

1. 在非唯一索引的情况下，删除一条存在的记录是有gap锁，锁住记录本身和记录两边的gap；
2. 在唯一索引和主键的情况下删除一条存在的记录，因为都是唯一值，进行删除的时候，是不会有gap存在；
3. 非唯一索引，唯一索引和主键在删除一条不存在的记录，均会在这个区间加gap锁；
4. 通过非唯一索引和唯一索引去删除一条标记为删除的记录的时候，都会请求该记录的行锁，同时锁住记录之前的gap；
5. RC情况下是没有gap锁的，除了遇到唯一键冲突的情况，如插入唯一键冲突。

# 查看死锁

## 查看锁状态

1、查看事务锁等待状态情况

select \* from information\_schema.innodb\_locks;

select \* from information\_schema.innodb\_lock\_waits;

select \* from information\_schema.innodb\_trx;

下面的查询可以得到当前状况下数据库的等待情况:

select r.trx\_id wait\_trx\_id,

r.trx\_mysql\_thread\_id wait\_thr\_id,

r.trx\_query wait\_query,

b.trx\_id block\_trx\_id,

b.trx\_mysql\_thread\_id block\_thrd\_id,

b.trx\_query block\_query

from information\_schema.innodb\_lock\_waits w

inner join information\_schema.innodb\_trx b on b.trx\_id = w.blocking\_trx\_id

inner join information\_schema.innodb\_trx r on r.trx\_id =w.requesting\_trx\_id

2、打开下列参数，获取更详细的事务和死锁信息。

innodb\_print\_all\_deadlocks = ON

innodb\_status\_output\_locks = ON

3、查看innodb状态(包含最近的死锁日志)

show engine innodb status;

## 分析死锁日志

# 处理死锁

死锁机制包含两部分:检测和处理。

把事务等待列表和锁等待信息列表通过事务信息进行wait-for graph检测，如果发现有闭环，则回滚undo log量少的事务；死锁检测本身也会算检测本身所需要的成本，以便应对检测超时导致的意外情况。

## 死锁检测

当InnoDB事务尝试获取(请求)加一个锁，并且需要等待时，InnoDB会进行死锁检测. 正常的流程如下:

1. InnoDB初始化一个事务，当事务尝试申请加锁，并且需要等待时(wait\_lock)，innodb会开始进行死锁检测(deadlock\_mark)。
2. 进入到lock\_deadlock\_check\_and\_resolve()函数进行检测死锁和解决死锁。
3. 检测死锁过程中，是由计数器来进行限制次数的，在等待wait-for graph检测过程中遇到超时或者超过阈值，则停止检测。
4. 死锁检测的逻辑之一是等待图的处理过程，如果通过锁的信息和事务等待链构造出一个图，如果图中出现回路，就认为发生了死锁。
5. 死锁的回滚，内部代码的处理逻辑之一是比较undo的数量，回滚undo数量少的事务。

## 处理死锁

《数据库系统实现》里面提到的死锁处理:

超时死锁检测:当存在死锁时，想所有事务都能同时继续执行通常是不可能的，因此，至少一个事务必须中止并重新开始。超时是最直接的办法，对超出活跃时间的事务进行限制和回滚。

等待图：等待图的实现，是可以表明哪些事务在等待其他事务持有的锁，可以在数据库的死锁检测里面加上这个机制来进行检测是否有环的形成。

通过元素排序预防死锁:这个想法很美好，但现实很残酷,通常都是发现死锁后才去想办法解决死锁的原因。

通过时间戳检测死锁:对每个事务都分配一个时间戳，根据时间戳来进行回滚策略。

# 避免死锁

1. 事务隔离级别使用read committed和binlog\_format=row，避免RR模式带来的gap锁竞争。
2. 合理的设计索引，区分度高的列放到组合索引前列，使业务sql尽可能通过索引定位更少的行，减少锁竞争。
3. 调整业务逻辑SQL执行顺序，避免update/delete长时间持有锁的SQL在事务前面，(该优化视情况而定)。
4. 选择合理的事务大小，小事务发生锁冲突的几率也更小。
5. 5.7.15 版本之后提供了新的功能innodb\_deadlock\_detect参数,可以关闭死锁检测，提高并发TPS，但是要注意设置锁等待时间innodb\_lock\_wait\_timeout。

# 案例一

## 问题初现

在某天下午，突然系统报警，抛出个异常：

仔细一看好像是事务回滚异常，写着的是因为死锁回滚，原来是个死锁问题，由于我对Mysql锁还是有一定了解的，于是开始主动排查这个问题。

首先在数据库中查找Innodb Status，在Innodb Status中会记录上一次死锁的信息,输入下面命令：

SHOW ENGINE INNODB STATUS

死锁信息如下,sql信息进行了简单处理：

------------------------

LATEST DETECTED DEADLOCK

------------------------

2019-02-22 15:10:56 0x7eec2f468700

\*\*\* (1) TRANSACTION:

TRANSACTION 2660206487, ACTIVE 0 sec starting index read

mysql tables in use 1, locked 1

LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)

MySQL thread id 31261312, OS thread handle 139554322093824, query id 11624975750 10.23.134.92 erp\_crm\_\_6f73 updating

/\*id:3637ba36\*/UPDATE tenant\_config SET

open\_card\_point = 0

where tenant\_id = 123

\*\*\* (1) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 1322 page no 534 n bits 960 index uidx\_tenant of table `erp\_crm\_member\_plan`.`tenant\_config` trx id 2660206487 lock\_mode X locks rec but not gap waiting

\*\*\* (2) TRANSACTION:

TRANSACTION 2660206486, ACTIVE 0 sec starting index read

mysql tables in use 1, locked 1

3 lock struct(s), heap size 1136, 2 row lock(s)

MySQL thread id 31261311, OS thread handle 139552870532864, query id 11624975758 10.23.134.92 erp\_crm\_\_6f73 updating

/\*id:3637ba36\*/UPDATE tenant\_config SET

open\_card\_point = 0

where tenant\_id = 123

\*\*\* (2) HOLDS THE LOCK(S):

RECORD LOCKS space id 1322 page no 534 n bits 960 index uidx\_tenant of table `erp\_crm\_member\_plan`.`tenant\_config` trx id 2660206486 lock mode S

\*\*\* (2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 1322 page no 534 n bits 960 index uidx\_tenant of table `erp\_crm\_member\_plan`.`tenant\_config` trx id 2660206486 lock\_mode X locks rec but not gap waiting

\*\*\* WE ROLL BACK TRANSACTION (1)

------------

给大家简单的分析解释一下这段死锁日志，事务1执行Update语句的时候需要获取uidx\_tenant这个索引再where条件上的X锁(行锁)，事务2执行同样的Update语句，也在uidx\_tenant上面想要获取X锁(行锁)，然后就出现了死锁，回滚了事务1。当时我就很懵逼，回想了一下死锁产生的必要条件:

1、互斥。

2、请求与保持条件。

3、不剥夺条件。

4、循环等待。

从日志上来看事务1和事务2都是取争夺同一行的行锁，和以往的互相循环争夺锁有点不同，怎么看都无法满足循环等待条件。经过同事提醒，既然从死锁日志中不能进行排查，那么就只能从业务代码和业务日志从排查。这段代码的逻辑如下：

public int saveTenantConfig(PoiContext poiContext, TenantConfigDO tenantConfig) {

try {

return tenantConfigMapper.saveTenantConfig(poiContext.getTenantId(), poiContext.getPoiId(), tenantConfig);

} catch (DuplicateKeyException e) {

LOGGER.warn("[saveTenantConfig] 主键冲突，更新该记录。context:{}, config:{}", poiContext, tenantConfig);

return tenantConfigMapper.updateTenantConfig(poiContext.getTenantId(), tenantConfig);

}

}

这段代码的意思是保存一个配置文件，如果发生了唯一索引冲突那么就会进行更新，当然这里可能写得不是很规范，其实可以用

insert into ...

on duplicate key update

也可以达到同样的效果，但是就算用这个其实也会发生死锁。看了代码之后同事又给我发了当时业务日志,

可以看见这里有三条同时发生的日志，说明都发生了唯一索引冲突进入了更新的语句，然后发生的死锁。到这里答案终于稍微有点眉目了。

这个时候再看我们的表结构如下(做了简化处理)：

CREATE TABLE `tenant\_config` (

`id` bigint(21) NOT NULL AUTO\_INCREMENT,

`tenant\_id` int(11) NOT NULL,

`open\_card\_point` int(11) DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `uidx\_tenant` (`tenant\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 ROW\_FORMAT=COMPACT

我们的tenant\_id是用来做唯一索引，我们的插入和更新的where条件都是基于唯一索引来操作的。

UPDATE tenant\_config SET

open\_card\_point = 0

where tenant\_id = 123

到了这里感觉插入的时候对唯一索引加锁有关系，接下来我们进行下一步的深入剖析。

## 深入剖析

上面我们说有三个事务进入update语句，为了简化说明这里我们只需要两个事务同时进入update语句即可，下面的表格展示了我们整个的发生过程：

小提示:S锁是共享锁，X锁是互斥锁。一般来说X锁和S，X锁都互斥，S锁和S锁不互斥。

我们从上面的流程中看见发生这个死锁的关键需要获取S锁，为什么我们再插入的时候需要获取S锁呢？因为我们需要检测唯一索引？在RR隔离级别下如果要读取那么就是当前读,那么其实就需要加上S锁。这里发现唯一键已经存在，这个时候执行update就会被两个事务的S锁互相阻塞，从而形成上面的循环等待条件。

小提示: 在MVCC中，当前读和快照读的区别:当前读每次需要加锁（可以使共享锁或者互斥锁）获取到最新的数据，而快照读是读取的是这个事务开始的时候那个快照，这个是通过undo log去进行实现的。

这个就是整个死锁的原因，能出现这种死锁的还有一个情况，就是同一时间来三个插入操作，其中先插入的那个事务如果最后回滚了，其余两个事务也会出现这种死锁。

## 解决方案

这里的核心问题是需要把S锁给干掉，这里有三个可供参考的解决方案:

将RR隔离级别，降低成RC隔离级别。这里RC隔离级别会用快照读，从而不会加S锁。

再插入的时候使用select \* for update,加X锁，从而不会加S锁。

可以提前加上分布式锁，可以利用Redis,或者ZK等等，分布式锁可以参考我的这篇文章。聊聊分布式锁

第一种方法不太现实，毕竟隔离级别不能轻易的修改。第三种方法又比较麻烦。所以第二种方法是我们最后确定的。

# 案例二

# 总结