链滴

MySQL InnoDB 如何保证事务特性

作者：jianzh5

原文链接：https://ld246.com/article/1571650979082

来源网站：链滴

如果有人问你“数据库事务有哪些特性”？你可能会很快回答出原子性、一致性、隔离性、持久性即CID特性。那么你知道InnoDB如何保证这些事务特性的吗？如果知道的话这篇文章就可以直接跳过不啦(#^.^#)

先说结论：

- redo log重做日志用来保证事务的持久性
- undo log回滚日志保证事务的原子性
- undo log+redo log保证事务的一致性
- 锁（共享、排他）用来保证事务的隔离性

重做日志 redo log

重做日志 redo log 分为两部分：一部分是内存中的重做日志缓冲（redo log buffer），是易丢失的二部分是重做日志文件（redo log file），是持久的。InnoDB通过Force Log at Commit机制来实持久性，当commit时，必须先将事务的所有日志写到重做日志文件进行持久化，待commit操作完成算完成。

InnoDB在下面情况下会将重做日志缓冲的内容写入重做日志文件：

- master thread 每一秒将重做日志缓冲刷新到重做日志文件;
- 每个事务提交时
- 当重做日志缓冲池剩余空间小于1/2时

为了确保每次日志都写入重做日志文件，在每次将日志缓冲写入重做日志文件后，InnoDB存储引擎需要调用一次fsync（刷盘）操作。但这也不是绝对的。用户可以通过修改innodb_flush_log_at_trx_commoit参数来控制重做日志刷新到磁盘的策略，这个可以作为大量事务提交时的优化点。

- 1参数默认值，表示事务提交时必须调用一次fsync操作。

● 0表示事务提交时，重做日志缓存并不立即写入重做日志文件，而是随着Master Thread的间隔进行fync操作。

● 2表示事务提交时将重做日志写入重做日志文件，但仅写入文件系统的缓存中，不进行fsync操作。

fsync的效率取决于磁盘的性能，因此磁盘的性能决定了事务提交的性能，也就是数据库的性能。**所如果有人问你如何优化Mysql数据库的时候别忘了有硬件这一条，让他们提升硬盘配置，换SSD固态盘**

重做日志都是以512字节进行存储的，称之为重做日志块，与磁盘扇区大小一致，这意味着重做日志写入可以保证原子性，不需要doublewrite技术。它有以下3个特性：

● 重做日志是在InnoDB层产生的

● 重做日志是物理格式日志，记录的是对每个页的修改

● 重做日志在事务进行中不断被写入，而且是顺序写入

回滚日志 undo log

为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地称为Undo Log），然后进行数据的修改。如果出现了错误或者用户执行了 ROLLBACK语句，系统可利用Undo Log中的备份将数据恢复到事务开始之前的状态。

undo log实现多版本并发控制（MVCC）来辅助保证事务的隔离性。

回滚日志不同于重做日志，它是逻辑日志，对数据库的修改都逻辑的取消了。当事务回滚时，它实际做的是与先前相反的工作。对于每个INSERT，InnoDB存储引擎都会完成一个DELETE；对于每个UPDTE，InnoDB存储引擎都会执行一个相反的UPDATE。

事务提交后并不能马上删除undo log，这是因为可能还有其他事务需要通过undo log 来得到行记录前的版本。故事务提交时将undo log 放入一个链表中，是否可以删除undo log 根据操作不同分以下种情况：

● Insert undo log： insert操作的记录，只对事务本身可见，对其他事务不可见(这是事务隔离性的求),故该undo log可以在事务提交后直接删除。不需要进行 purge操作。

● update undo log：记录的是对 delete和 update操作产生的 undo log。该undo log可能需要提MVCC机制，因此不能在事务提交时就进行删除。提交时放入undo log链表,等待 purge线程进行最的删除。

锁

事务的隔离性的实现原理就是锁，因而隔离性也可以称为并发控制、锁等。事务的隔离性要求每个读事务的对象对其他事务的操作对象能互相分离。再者，比如操作缓冲池中的LRU列表，删除，添加、动LRU列表中的元素，为了保证一致性那么就要锁的介入。

锁的类型

InnoDB主要有2种锁：行级锁，意向锁

行级锁：

● 共享锁（读锁 S），允许事务读一行数据。事务拿到某一行记录的共享S锁，才可以读取这一行，阻止别的事务对其添加X锁。共享锁的目的是提高读读并发。

● 排它锁（写锁 X），允许事务删除一行数据或者更新一行数据。事务拿到某一行记录的排它X锁，

可以修改或者删除这一行。排他锁的目的是为了保证数据的一致性。

行级锁中，除了S和S兼容，其他都不兼容。

意向锁：

● 意向共享锁（读锁 IS ），事务想要获取一张表的几行数据的共享锁，事务在给一个数据行加共享前必须先取得该表的IS锁。

● 意向排他锁（写锁 IX），事务想要获取一张表中几行数据的排它锁，事务在给一个数据行加排他锁必须先取得该表的IX锁。

解释一下意向锁

The main purpose of IX and IS locks is to show that someone is locking a row, or going to loc a row in the table.

意向锁的主要用途是为了表达某个事务正在锁定一行或者将要锁定一行数据。e.g:事务A要对一行记录进行上X锁，那么InnoDB会先申请表的IX锁，再锁定记录r的X锁。在事务A完成之前，事务B想要来全表操作，此时直接在表级别的IX就告诉事务B需要等待而不需要在表上判断每一行是否有锁。意向它锁存在的价值在于节约InnoDB对于锁的定位和处理性能。另外注意了，除了全表扫描以外意向锁不会阻塞。

锁的算法

InnoDB有三种行锁的算法：

● Record Lock：单个行记录上的锁

● Gap Lock：间隙锁，锁定一个范围，而非记录本身

● Next-Key Lock：结合Gap Lock和Record Lock，锁定一个范围，并且锁定记录本身。主要解决的题是REPEATABLE READ隔离级别下的幻读。可以参考文章了解事务隔离级别的相关知识点。

这里主要讲一下Next-Key Lock，利用Next-key Lock锁定的不是单个值而是一个范围，他的目的就为了阻止多个事务将记录插入到同一范围内从而导致幻读。

注意了，如果走唯一索引，那么Next-Key Lock会降级为Record Lock，即仅锁住索引本身，而不范围。也就是说Next-Key Lock前置条件为事务隔离级别为RR且查询的索引走的非唯一索引、主键引。

下面我们用个例子详细说一下。

首先建立一张表：

```
CREATE TABLE T (id int ,f_id int,PRIMARY KEY (id), KEY(f_id)) ENGINE=InnoDB DEFAULT CHAR
ET=utf8
insert into T SELECT 1,1;
insert into T SELECT 3,1;
insert into T SELECT 5,3;
insert into T SELECT 7,6;
insert into T SELECT 10,8;
```

事务A执行如下语句：

```
SELECT * FROM T WHERE f_id = 3 FOR UPDATE
```

这时SQL语句走非唯一索引，因此使用Next-Key Locking加锁，并且有2个索引，其需要分别进行锁。

对于聚集索引，其仅对id等于5的索引加上Record Lock。而对于辅助索引，其加上Next-Key Lock，定了范围（1,3），特别需要注意的是，InnoDB存储引擎还会对辅助索引下一个键值加上Gap Lock即范围（3.6）的锁。

所以如果在新session中执行如下语句都会报错[Err] 1205 - Lock wait timeout exceeded; try restarting transaction：

```
select * from T where id = 5 lock in share MODE -- 不能执行，因为事务A已经给id=5的值加上了锁，执行会被阻塞
INSERT INTO T SELECT 4,2  -- 不能执行，辅助索引的值为2，在（1,3）的范围内，执行阻塞
INSERT INTO T SELECT 6,5  -- 不能执行，gap锁会锁住（3,6）的范围，执行阻塞
```

此时想象一下，事务A锁定了f_id =5 的记录， 正常会有个gap lock，锁住（5,6），那么如果没有（56）的gap锁，那么用户可以插入索引 f_id 为5的记录，这样事务A再次查询就会返回一个不同的记录也就导致了幻读的产生。

同理，如果我们事务A执行的是select * from T where f_id = 10 FOR UPDATE,在表里查不到数据，是基于Next-Key Lock会锁住（8，+∞），我们执行INSERT INTO T SELECT 6,11是无法插入成功的这就从根本上解决了幻读问题。