MySQL事务隔离级别和MVCC

原创 小孩子4919 我们都是小青蛙 2019-03-28

事前准备

为了故事的顺利发展，我们需要创建一个表：

```
CREATE TABLE t (
    id INT PRIMARY KEY,
    c VARCHAR(100)
) Engine=InnoDB CHARSET=utf8;
```

然后向这个表里插入一条数据：

```
INSERT INTO t VALUES(1, '刘备');
```

现在表里的数据就是这样的：

```
mysql> SELECT * FROM t;
+----+--------+
| id | c      |
+----+--------+
|  1 | 刘备   |
+----+--------+
1 row in set (0.01 sec)
```

隔离级别

MySQL 是一个服务器 / 客户端架构的软件，对于同一个服务器来说，可以有若干个客户端与之连接，每个客户端与服务器连接上之后，就可以称之为一个会话（ Session ）。我们可以同时在不同的会话里输入各种语句，这些语句可以作为事务的一部分进行处理。不同的会话可以同时发送请求，也就是说服务器可能同时在处理多个事务，这样子就会导致不同的事务可能同时访问到相同的记录。我们前边说过事务有一个特性称之为 隔离性 ，理论上在某个事务对某个数据进行访问时，其他事务应该进行排队，当该事务提交之后，其他事务才可以继续访问这个数据。但是这样子的话对性能影响太大，所以设计数据库的大叔提出了各种 隔离级别 ，来最大限度的提升系统并发处理事务的能力，但是这也是以牺牲一定的 隔离性 来达到的。

未提交读（READ UNCOMMITTED）

如果一个事务读到了另一个未提交事务修改过的数据，那么这种 隔离级别 就称之为 未提交读 （英文名：READ UNCOMMITTED ），示意图如下：

如上图， Session A 和 Session B 各开启了一个事务， Session B 中的事务先将 id 为 1 的记录的列 c 更新为 '关羽' ，然后 Session A 中的事务再去查询这条 id 为 1 的记录，那么在 未提交读 的隔离级别下，查询结果就是 '关羽' ，也就是说某个事务读到了另一个未提交事务修改过的记录。但是如果 Session B 中的事务稍后进行了回滚，那么 Session A 中的事务相当于读到了一个不存在的数据，这种现象就称之为 脏读 ，就像这个样子：

脏读 违背了现实世界的业务含义，所以这种 READ UNCOMMITTED 算是十分不安全的一种 隔离级别 。

已提交读（READ COMMITTED）

如果一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那么这种 隔离级别 就称之为 已提交读 （英文名： READ COMMITTED ），如图所示：

从图中可以看到，第4步时，由于 Session B 中的事务尚未提交，所以 Session A 中的事务查询得到的结果只是 '刘备' ，而第6步时，由于 Session B 中的事务已经提交，所以 Session B 中的事务查询得到的结果就是 '关羽' 了。

对于某个处在在 已提交读 隔离级别下的事务来说，只要其他事务修改了某个数据的值，并且之后提交了，那么该事务就会读到该数据的最新值，比方说：

我们在 Session B 中提交了几个隐式事务，这些事务都修改了 id 为 1 的记录的列c的值，每次事务提交之后， Session A 中的事务都可以查看到最新的值。这种现象也被称之为 不可重复读 。

可重复读（REPEATABLE READ）

在一些业务场景中，一个事务只能读到另一个已经提交的事务修改过的数据，但是第一次读过某条记录后，即使其他事务修改了该记录的值并且提交，该事务之后再读该条记录时，读到的仍是第一次读到的值，而不是每次都读到不同的数据。那么这种 隔离级别 就称之为 可重复读 （英文名： REPEATABLE READ ），如图所示：

从图中可以看出来， Session A 中的事务在第一次读取 id 为 1 的记录时，列 c 的值为 '刘备' ，之后虽然 Session B 中隐式提交了多个事务，每个事务都修改了这条记录，但是 Session A 中的事务读到的列 c 的值仍为 '刘备' ，与第一次读取的值是相同的。

串行化（SERIALIZABLE）

以上3种隔离级别都允许对同一条记录进行 读-读 、 读-写 、 写-读 的并发操作，如果我们不允许 读-写 、 写-读 的并发操作，可以使用 SERIALIZABLE 隔离级别，示意图如下：

如图所示，当 Session B 中的事务更新了 id 为 1 的记录后，之后 Session A 中的事务再去访问这条记录时就被卡住了，直到 Session B 中的事务提交之后， Session A 中的事务才可以获取到查询结果。

版本链

对于使用 InnoDB 存储引擎的表来说，它的聚簇索引记录中都包含两个必要的隐藏列（ row_id 并不是必要的，我们创建的表中有主键或者非NULL唯一键时都不会包含 row_id 列）：

- trx_id：每次对某条聚簇索引记录进行改动时，都会把对应的事务id赋值给 trx_id 隐藏列。
- roll_pointer：每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到 undo日志 中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

比方说我们的表 t 现在只包含一条记录：

```
mysql> SELECT * FROM t;
+----+--------+
| id | c      |
+----+--------+
|  1 | 刘备   |
+----+--------+
1 row in set (0.01 sec)
```

假设插入该记录的事务id为 80 ，那么此刻该条记录的示意图如下所示：

假设之后两个 id 分别为 100 、 200 的事务对这条记录进行 UPDATE 操作，操作流程如下：

> 小贴士： 能不能在两个事务中交叉更新同一条记录呢？哈哈，这是不可以滴，第一个事务更新了某条记录后，就会给这条记录加锁，另一个事务再次更新时就需要等待第一个事务提交了，把锁释放之后才可以继续更新。本篇文章不是讨论锁的，有关锁的更多细节我们之后再说。

每次对记录进行改动，都会记录一条 undo日志 ，每条 undo日志 也都有一个 roll_pointer 属性（ INSERT 操作对应的 undo日志 没有该属性，因为该记录并没有更早的版本），可以将这些 undo日志 都连起来，串成一个链表，所以现在的情况就像下图一样：

对该记录每次更新后，都会将旧值放到一条 undo日志 中，就算是该记录的一个旧版本，随着更新次数的增多，所有的版本都会被 roll_pointer 属性连接成一个链表，我们把这个链表称之为 版本链 ，版本链的头节点就是当前记录最新的值。另外，每个版本中还包含生成该版本时对应的事务id，这个信息很重要，我们稍后就会用到。

ReadView

对于使用 READ UNCOMMITTED 隔离级别的事务来说，直接读取记录的最新版本就好了，对于使用 SERIALIZABLE 隔离级别的事务来说，使用加锁的方式来访问记录。对于使用 READ COMMITTED 和 REPEATABLE READ 隔离级别的事务来说，就需要用到我们上边所说的 版本链 了，核心问题就是：需要判断一下版本链中的哪个版本是当前事务可见的。所以设计 InnoDB 的大叔提出了一个 ReadView 的概念，这个 ReadView 中主要包含当前系统中还有哪些活跃的读写事务，把它们的事务id放到一个列表中，我们把这个列表命名为为 m_ids 。这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

- 如果被访问版本的 trx_id 属性值小于 m_ids 列表中最小的事务id，表明生成该版本的事务在生成 ReadView 前已经提交，所以该版本可以被当前事务访问。

- 如果被访问版本的 trx_id 属性值大于 m_ids 列表中最大的事务id，表明生成该版本的事务在生成 ReadView 后才生成，所以该版本不可以被当前事务访问。

- 如果被访问版本的 trx_id 属性值在 m_ids 列表中最大的事务id和最小事务id之间，那就需要判断一下 trx_id 属性值是不是在 m_ids 列表中，如果在，说明创建 ReadView 时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建 ReadView 时生成该版本的事务已经被提交，该版本可以被访问。

如果某个版本的数据对当前事务不可见的话，那就顺着版本链找到下一个版本的数据，继续按照上边的步骤判断可见性，依此类推，直到版本链中的最后一个版本，如果最后一个版本也不可见的话，那么就意味着该条记录对该事务不可见，查询结果就不包含该记录。
在 MySQL 中， READ COMMITTED 和 REPEATABLE READ 隔离级别的的一个非常大的区别就是它们生成 ReadView 的时机不同，我们来看一下。

READ COMMITTED ---- 每次读取数据前都生成一个ReadView

比方说现在系统里有两个 id 分别为 100 、 200 的事务在执行：

```
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;

# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...
```

> 小贴士： 事务执行过程中，只有在第一次真正修改记录时（比如使用INSERT、DELETE、UPDATE语句），才会被分配一个单独的事务id，这个事务id是递增的。

此刻，表 t 中 id 为 1 的记录得到的版本链表如下所示：

假设现在有一个使用 READ COMMITTED 隔离级别的事务开始执行：

```
# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
```

这个 SELECT1 的执行过程如下：

- 在执行 SELECT 语句时会先生成一个 ReadView ， ReadView 的 m_ids 列表的内容就是 [100, 200] 。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 c 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 c 的内容是 '关羽' ，该版本的 trx_id 值也为 100 ，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 c 的内容是 '刘备' ，该版本的 trx_id 值为 80 ，小于 m_ids 列表中最小的事务 id 100 ，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 c 为 '刘备' 的记录。

之后，我们把事务id为 100 的事务提交一下，就像这样：

```
# Transaction 100
BEGIN;
```

```
UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;

COMMIT;
```

然后再到事务id为 200 的事务中更新一下表 t 中 id 为1的记录：

```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE t SET c = '赵云' WHERE id = 1;

UPDATE t SET c = '诸葛亮' WHERE id = 1;
```

此刻，表 t 中 id 为 1 的记录的版本链就长这样：

然后再到刚才使用 READ COMMITTED 隔离级别的事务中继续查找这个id为 1 的记录，如下：

```
# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200均未提交
```

```
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'张飞'
```

这个 SELECT2 的执行过程如下：

- 在执行 SELECT 语句时会先生成一个 ReadView ， ReadView 的 m_ids 列表的内容就是 [200] （事务id为 100 的那个事务已经提交了，所以生成快照时就没有它了）。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 c 的内容是 '诸葛亮' ，该版本的 trx_id 值为 200 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 c 的内容是 '赵云' ，该版本的 trx_id 值为 200 ，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 c 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，比 m_ids 列表中最小的事务 id 200 还要小，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 c 为 '张飞' 的记录。

以此类推，如果之后事务id为 200 的记录也提交了，再此在使用 READ COMMITTED 隔离级别的事务中查询表 t 中 id 值为 1 的记录时，得到的结果就是 '诸葛亮' 了，具体流程我们就不分析了。总结一下就是：使用READ COMMITTED隔离级别的事务在每次查询开始时都会生成一个独立的ReadView。

REPEATABLE READ ---在第一次读取数据时生成一个ReadView

对于使用 REPEATABLE READ 隔离级别的事务来说，只会在第一次执行查询语句时生成一个 ReadView ，之后的查询就不会重复生成了。我们还是用例子看一下是什么效果。

比方说现在系统里有两个 id 分别为 100 、 200 的事务在执行：

```
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;
```

```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...
```

此刻，表 t 中 id 为 1 的记录得到的版本链表如下所示：

假设现在有一个使用 REPEATABLE READ 隔离级别的事务开始执行：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
```

这个 SELECT1 的执行过程如下：

- 在执行 SELECT 语句时会先生成一个 ReadView ， ReadView 的 m_ids 列表的内容就是 [100, 200] 。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 c 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。

- 下一个版本的列 `c` 的内容是 `'关羽'` ，该版本的 `trx_id` 值也为 `100` ，也在 `m_ids` 列表内，所以也不符合要求，继续跳到下一个版本。

- 下一个版本的列 `c` 的内容是 `'刘备'` ，该版本的 `trx_id` 值为 `80` ，小于 `m_ids` 列表中最小的事务id `100` ，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 `c` 为 `'刘备'` 的记录。

之后，我们把事务id为 `100` 的事务提交一下，就像这样：

```
# Transaction 100
BEGIN;

UPDATE t SET c = '关羽' WHERE id = 1;

UPDATE t SET c = '张飞' WHERE id = 1;

COMMIT;
```

然后再到事务id为 `200` 的事务中更新一下表 `t` 中 `id` 为1的记录：

```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE t SET c = '赵云' WHERE id = 1;

UPDATE t SET c = '诸葛亮' WHERE id = 1;
```

此刻，表 `t` 中 `id` 为 `1` 的记录的版本链就长这样：

然后再到刚才使用 REPEATABLE READ 隔离级别的事务中继续查找这个id为 1 的记录，如下：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200均未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM t WHERE id = 1; # 得到的列c的值仍为'刘备'
```

这个 SELECT2 的执行过程如下：

- 因为之前已经生成过 ReadView 了，所以此时直接复用之前的 ReadView ，之前的 ReadView 中的 m_ids 列表就是 [100, 200] 。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 c 的内容是 '诸葛亮' ，该版本的 trx_id 值为 200 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 c 的内容是 '赵云' ，该版本的 trx_id 值为 200 ，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。

- 下一个版本的列 c 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，而 m_ids 列表中是包含值为 100 的事务id的，所以该版本也不符合要求，同理下一个列 c 的内容是 '关羽' 的版本也不符合要求。继续跳到下一个版本。

- 下一个版本的列 c 的内容是 '刘备' ，该版本的 trx_id 值为 80 ， 80 小于 m_ids 列表中最小的事务id 100 ，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 c 为 '刘备' 的记录。

也就是说两次 SELECT 查询得到的结果是重复的，记录的列 c 值都是 '刘备' ，这就是 可重复读 的含义。如果我们之后再把事务id为 200 的记录提交了，之后再到刚才使用 REPEATABLE READ 隔离级别的事务中继续查找这个id为 1 的记录，得到的结果还是 '刘备' ，具体执行过程大家可以自己分析一下。

MVCC总结

从上边的描述中我们可以看出来，所谓的MVCC（Multi-Version Concurrency Control ，多版本并发控制）指的就是在使用 READ COMMITTD 、 REPEATABLE READ 这两种隔离级别的事务在执行普通的 SEELCT 操作时访问记录的版本链的过程，这样子可以使不同事务的 读-写 、 写-读 操作并发执行，从而提升系统性能。 READ COMMITTD 、 REPEATABLE READ 这两个隔离级别的一个很大不同就是生成 ReadView 的时机不同， READ COMMITTD 在每一次进行普通 SELECT 操作前都会生成一个 ReadView ，而 REPEATABLE READ 只在第一次进行普通 SELECT 操作前生成一个 ReadView ，之后的查询操作都重复这个 ReadView 就好了。

题外话

码字不易，有帮助的话可以转发一波，公众号排版不太好，想看更美观的排版可以点击原文。

各位随手点个好看呗

阅读原文