链滴

脏读、幻读和不可重复读

作者：wangning1018
原文链接：https://ld246.com/article/1578385783859
来源网站：链滴

一、引言

脏读、不可重复读和幻读是数据库中由于**并发访问**导致的数据读取问题。当多个事务同时进行时可以过修改数据库事务的隔离级别来处理这三个问题。

二、问题解释

1、脏读（读取未提交的数据）

脏读又称无效数据的读出，是指在数据库访问中，事务 A 对一个值做修改，事务 B 读取这个值，但由于某种原因事务 A 回滚撤销了对这个值得修改，这就导致事务 B 读取到的值是无效数据。

2、不可重复读（前后数据多次读取，结果集内容不一致）

不可重复读即当事务 A 按照查询条件得到了一个结果集，这时事务 B 对事务 A 查询的结果集数据做修改操作，之后事务 A 为了数据校验继续按照之前的查询条件得到的结果集**与前一次查询不同**，导致可重复读取原始数据。

3、幻读（前后数据多次读取，结果集数量不一致）

幻读是指当事务 A 按照查询条件得到了一个结果集，这时事务 B 对事务 A 查询的结果集数据做新增作，之后事务 A 继续按照之前的查询条件得到的结果集平白无故**多了几条数据**，好像出现了幻觉一样。

三、事务隔离

在并发条件下会出现上述问题，如何着手解决他们保证我们程序运行的正确性是非常重要的。数据库供了 **Read uncommitted 、Read committed 、Repeatable read 、Serializable** 四种事务隔离别来解决脏读、幻读和不可重复读问题，同时容易想到，可以通过加锁的方式实现事务隔离。

在数据库的增删改查操作中，insert 、delete 、update 都会加排他锁，**排它锁会阻止其他事务对其锁的数据加任何类型的锁**。而 select 只有显示声明才会加锁。

- Read uncommitted

读未提交，说的是一个事务可以读取到另一个事务未提交的数据修改。

读若不显式声明是不加锁的，可以直接读取到另一个事务对数据的操作，没有避免脏读、不可重复读幻读。

- Read committed

读已提交，说的是一个事务只能读取到另一个事务已经提交的数据修改。

很明显，这种隔离级别避免了脏读，但是可能会出现不可重复读、幻读。

- Repeatable read

可重复读，保证了同一事务下多次读取相同的数据返回的结果集是一样的。

这种隔离级别解决了脏读和不可重复读问题，但是扔有可能出现幻读。

- Serializable

串行化，对同一数据的读写全加锁，即对同一数据的读写全是互斥了，数据可靠行很强，但是并发性不忍直视。

这种隔离级别虽然解决了上述三个问题，但是牺牲了性能。

总结如下表： √ 代表可能出现，× 代表不会出现。

隔离级别	**脏读**	**不可重复读**	**读**
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

四、MySQL 事务隔离级别的实现

在 MySQL 中只有 InnoDB 存储引擎支持事务，但是在日常使用 MySQL 时我们好像没有怎么关心过述三个问题啊...

原因很简单，MySQL 默认 **Repeatable read** 隔离级别，使用了 MVCC 技术，并且解决了幻读问题。

MVCC

MVCC 全名多版本并发控制，使用它可以保证 InnoDB 存储引擎下读操作的一致性。使用 MVCC 可查询被另一个事务修改的行数据，并且可以查看这些行被更新之前的数据，值得注意的是**使用 MVCC 增加了多事务的并发性能，但是并没有解决幻读问题**。

1、原理

MVCC 是通过保存数据在某个时间点的快照来实现的。也就是说在同一个事务的生命周期中，数据的照始终是相同的；而在多个事务中，由于事务的时间点很可能不相同，数据的快照也不尽相同。

2、实现细节

- 每行数据都存在一个版本，每次数据更新时都更新该版本。
- 修改时Copy出当前版本随意修改，各个事务之间互不干扰。
- 保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃copy（rollback）。

通过上面特点我们可以看出，MVCC 其实就是类似乐观锁的一种实现。

3、InnoDB 中 MVCC 实现

在 InnoDB 中为每行增加两个隐藏的字段，分别是该行数据**创建时的版本号**和**删除时的版本号**，这里版本号是系统版本号（可以简单理解为事务的 ID），每开始一个新的事务，系统版本号就自动递增，为事务的 ID 。通常这两个版本号分别叫做创建时间和删除时间。

下面通过具体的例子来帮助理解 InnoDB 中 MVCC 实现，

首先创建一个表：

```
create table info(
id int primary key auto_increment,
name varchar(20));
```

INSERT

InnoDB 为新插入的每一行保存当前系统版本号作为版本号。现在假设事务的版本号从 1 开始。

第一个事务 ID为1；

```
start transaction;
insert into info values(NULL,'a');
insert into info values(NULL,'b');
insert into info values(NULL,'c');
commit;
```

对应在数据中的表如下(后面两列是隐藏列,也就是版本号)

id	name	创建版本(事务ID)	除版本(事务ID)
1	a	1	undefined
2	b	1	undefined
3	c	1	undefined

SELECT

InnoDB 会根据下面两个条件检查每行记录:

- 只会查找版本 **早于当前事务版本的数据行**（行的系统版本号小于或等于事务的系统版本号），这样以确保事务读取的行，要么是**在事务开始前已经存在的**，要么是**事务自身插入或者修改过的。**

● **行的删除版本要么未定义,要么大于当前事务版本号**,这可以确保事务读取到的行，在**事务开始之前被删除。**

只有 a, b 同时满足的记录，才能返回作为查询结果.

DELETE

InnoDB会为删除的每一行保存当前系统的版本号(事务的ID)作为删除标识.

看下面的具体例子分析:

第二个事务 ID为2;

```
start transaction;
select * from info;  //(1)
select * from info;  //(2)
commit;
```

● 假设1

假设在执行这个事务 ID 为 2 的过程中，刚执行到 (1) ,这时,有另一个事务 ID 为 3 往这个表里插入了条数据;

第三个事务ID为3;

```
start transaction;
insert into info values(NULL,'d');
commit;
```

这时表中的数据如下:

id	**name**	**创建版本(事务ID)**	**除版本(事务ID)**
1	a	1	undefined
2	b	1	undefined
3	c	1	undefined
4	d	3	undefined

然后接着执行 **事务2** 中的 (2) ， 由于 id=4 的数据的创建时间(事务 ID 为 3),执行当前事务的 ID 为 2 ，而 InnoDB 只会查找事务 ID 小于等于当前事务 ID 的数据行,所以 id=4 的数据行并不会在执行 **务2** 中的 (2) 被检索出来,在 **事务2* *中的两条 select 语句检索出来的数据都只会如下表:

id	**name**	**创建版本(事务ID)**	**除版本(事务ID)**
1	a	1	undefined
2	b	1	undefined
3	c	1	undefined

● 假设2

假设在执行这个事务 ID 为 2 的过程中，刚执行到 (1) ,假设事务执行完 **事务3** 后，接着又执行了 **事务4** ;

第四个事务:

```
start  transaction;
delete from info where id=1;
commit;
```

此时数据库中的表数据如下:

id	name	创建版本(事务ID)	除版本(事务ID)
1	a	1	4
2	b	1	undefined
3	c	1	undefined
4	d	3	undefined

接着执行事务 ID 为 2 的 **事务(2)**，根据 SELECT 检索条件可以知道,它会检索创建时间(创建事务的 ID 小于当前事务 ID 的行和删除时间(删除事务的 ID)大于当前事务的行,而 id=4 的行上面已经说过，而 i =1 的行由于删除时间(删除事务的 ID)大于当前事务的 ID ， 所以 **事务2** 的 (2) select * from info 也会把 id=1 的数据检索出来。所以，**事务2** 中的两条 select 语句检索出来的数据都如下:

id	name	创建版本(事务ID)	除版本(事务ID)
1	a	1	4
2	b	1	undefined
3	c	1	undefined

UPDATE

InnoDB 执行 UPDATE，实际上是**新插入了一行记录**，并保存其创建时间为当前事务的 ID ， 同时保 当前事务 ID 到要 UPDATE 的行的删除时间。

- 假设3

假设在执行完 **事务2** 的 (1) 后又执行，其它用户执行了事务 3和 4，这时，又有一个用户对这张表执了 UPDATE 操作:

第五个事务:

```
start  transaction;
update info set name='b' where id=2;
commit;
```

根据update的更新原则：会生成新的一行，并在原来要修改的列的删除时间列上添加本事务ID，得表如下:

id	name	创建版本(事务ID)

除版本(事务ID)			
1	a	1	4
2	b	1	5
3	c	1	undefined
4	d	3	undefined
2	b	5	undefined

继续执行 **事务2** 的 (2)，根据 select 语句的检索条件，得到下表:

id	name	创建版本(事务ID)	除版本(事务ID)
1	a	1	4
2	b	1	5
3	c	1	undefined

还是和 **事务2** 中 (1) select 得到相同的结果。

 总结:

- SELECT

读取创建版本号小于或等于当前事务版本号，并且删除版本号为空或大于当前事务版本号的记录。如可以保证在事务在读取之前记录是存在的。

- INSERT

将当前事务的版本号保存至插入行的创建版本号。

- UPDATE

新插入一行，并以当前事务的版本号作为新行的创建版本号，同时将原记录行的删除版本号设置为当事务版本号。

- DELETE

将当前事务的版本号保存至行的删除版本号。

例子参考：https://blog.csdn.net/whoamiyang/article/details/51901888

4、 InnoDB 如何解决幻读问题

在 InnoDB 中分为**快照读**和**当前读**。快照读读的是数据的快照，也就是数据的历史版本；当前读就是的最新版本的数据，并且在读的时候加锁，其他事务都不能对当前行做修改。

- 快照读：简单的 select 操作，属于快照读，不加锁。

```
select * from table where ?;
```

- 当前读：特殊的读操作，插入、更新、删除操作，属于当前读，需要加锁。

```
select * from table where ? lock in share mode;
select * from table where ? for update;
```

```
insert into table values (...);
update table set ? where ?;
delete from table where ?;
```

对于上面当前读的语句，第一条读取记录加共享锁，其他的全部加排它锁。

也就是说在做数据的修改操作时，都会使用当前读的方式，当前读是通过行锁和间隙锁控制的，此时加了排他锁的，所有其他的事务都不能动当前的事务，所以避免了出现幻读的可能。

而为了防止幻读，行锁和间隙锁扮演了重要角色，下面简单说一下：

- 行锁

字面意思简单理解对数据行加锁，注意 InnoDB 行锁是通过给索引上的索引项加锁来实现的，也就是说**有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！**

- 间隙锁

间隙锁就是用来为数据行之间的间隙来进行加锁。

举个例子：

```
select * from info where id > 5;
```

上面 SQL 中，其中 id 是主键，假设在一个 **事务 A** 中执行这个查询，第一次查询为一个 **结果集 1** 。做第二次查询时，另一个 **事务 B** 在 info 表进行了插入数据 7 和 10 的操作。在 **事务 A** 再次执行此询查询出 **结果集 2** 的时候，发现多了几条记录，如此便产生了幻读。

- 结果集1

```
6,8,9
```

- 结果集2

```
6,7,8,9,10
```

所以试想为了防止幻读，我们不但要现存的 id > 5 的数据行（6,8,9）上面加锁（行锁），还要在它的间隙加锁（间隙锁）。

我们以区间来表示要加锁对象：

(5,6]

(6,8]

(8,9]

(9,+∞)

其中区间的右闭即为要加的行锁，而区间的范围即是要加的间隙锁。

五、结语

关于脏读、不可重复读和幻读的理解便记录到这里了，因笔者水平有限，如有错误欢迎指正。