1.查看与设置事务隔离级别

在了解之前，需要先清楚事务的隔离级别怎样设置。

分别为当前会话的事务隔离级别，系统全局的事务隔离级别global。

mysql> select @@tx\_isolation;

+-----------------+

| @@tx\_isolation |

+-----------------+

| REPEATABLE-READ |

+-----------------+

1 row in set, 1 warning (0.00 sec)

mysql> select @@global.tx\_isolation;

+-----------------------+

| @@global.tx\_isolation |

+-----------------------+

| REPEATABLE-READ |

+-----------------------+

1 row in set, 1 warning (0.00 sec)

mysql> set session transaction isolation level read uncommitted;

Query OK, 0 rows affected (0.00 sec)

mysql> set global transaction isolation level read uncommitted;

Query OK, 0 rows affected (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

2.名词解释

2.1.脏读

一个事务读取到了其它事务还未提交的数据。

2.2.不可重复读

一个事务对同一行记录，两次读取到的结果不一样。

2.3.幻读

一个事务对同一个范围的记录，两次读取到的结果不一样。

3.四种隔离级别与具体实验

为了更好的看到差异性，实验将会使用两个终端来进行。

3.1.read uncommitted 未提交读

将事务隔离级别修改为read uncommitted，然后进行以下实验。

3.1.1.终端A先开启事务，然后查询user\_info表的数据

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 15 |

+----+-------+-----+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

3.1.2.终端B开启事务，更新user\_info表的数据，但是不commit

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> update user\_info set age=12 where id=1;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

1

2

3

4

5

6

3.1.3.终端A再次查询user\_info表的数据

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 12 |

+----+-------+-----+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

可以看到，此时查询出来的数据已经是修改过的了，虽然终端B还没有commit，但是在其它事务里已经读到了终端B对数据的修改，这就是脏读dirty read问题。

3.2.read committed 提交读

将事务隔离级别修改为read committed，然后进行以下实验。

3.2.1.终端A开启事务，查询user\_info的数据

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 15 |

+----+-------+-----+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

3.2.2.终端B开启事务，修改user\_info的数据，但是不commit

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> update user\_info set age=2 where id=1;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

1

2

3

4

5

6

3.2.3.终端A再次查询user\_info的数据，此时读取到的数据没有变化，脏读的问题被解决

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 15 |

+----+-------+-----+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

3.2.4.终端B执行commit

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

1

2

3.2.5.终端A再次查询user\_info的数据，此时读取到的数据发生了变化，同一个事务中，前后读取到的数据可能是变化的不一致的，这就是不可重复读的问题

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 2 |

+----+-------+-----+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

除了不可重复读问题之外，read committed隔离级别还面临着幻读的问题，以上的实验，终端B的update操作替换为insert向user\_info插入一条新数据然后commit，就会对终端A的事务造成幻读的问题。

3.3.repeatable read可重复读

可重复读是MySQL默认的事务隔离级别，它解决了不可重复读的问题，但初始的时候没有解决幻读的问题。

后来使用MVCC机制将幻读的问题也解决了，所以当前的可重复读隔离机制是在解决脏读、不可重复读、幻读问题的基础上，能够提供最大的并发性支持的隔离机制。

将事务隔离级别修改为repeatable read，然后进行以下实验。

3.3.1.1.终端A开启事务，查询user\_info表

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 8 |

| 2 | lucy | 16 |

| 3 | jeny | 14 |

+----+-------+-----+

3 rows in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

3.3.1.2.终端B开启事务，对user\_info表做update、insert、delete操作，但是不进行commit

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> update user\_info set age=22 where id=1;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> insert into user\_info(age,name) values(18,'lily');

Query OK, 1 row affected (0.00 sec)

mysql> delete from user\_info where id=2;

Query OK, 1 row affected (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

3.3.1.3.终端A再次查询user\_info表

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 8 |

| 2 | lucy | 16 |

| 3 | jeny | 14 |

+----+-------+-----+

3 rows in set (0.00 sec)

1

2

3

4

5

6

7

8

9

3.3.1.4.终端B进行commit

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

1

2

3.3.1.5.终端A再次查询user\_info

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 8 |

| 2 | lucy | 16 |

| 3 | jeny | 14 |

+----+-------+-----+

3 rows in set (0.00 sec)

1

2

3

4

5

6

7

8

9

3.3.1.6.终端A提交本次事务后，再查询user\_info

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from user\_info;

+----+-------+-----+

| id | name | age |

+----+-------+-----+

| 1 | testA | 22 |

| 3 | jeny | 14 |

| 4 | lily | 18 |

+----+-------+-----+

3 rows in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

可以看到，对于repeatable read隔离机制来讲，一个事务在执行过程中，无论其它的事务做了什么关于数据的update、insert、delete操作，都不会影响本事务，数据在事务执行的过程中始终是一致的。

其实repeatable read对幻读的解决并不彻底，在某些情况下仍然存在幻读的情况，看下面的例子：

3.3.2.1.终端A启动事务

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from a;

+------+

| id |

+------+

| 1 |

+------+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

3.3.2.2.终端B启动事务

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from a;

+------+

| id |

+------+

| 1 |

+------+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

3.3.2.3.终端A修改记录并提交

mysql> update a set id=id+1;

Query OK, 1 row affected (0.01 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> select \* from a;

+------+

| id |

+------+

| 2 |

+------+

1 row in set (0.00 sec)

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

3.3.2.4.终端B修改记录

mysql> select \* from a;

+------+

| id |

+------+

| 1 |

+------+

1 row in set (0.00 sec)

mysql> update a set id=id+2;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> select \* from a;

+------+

| id |

+------+

| 4 |

+------+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

按照终端B的事务逻辑来说，update操作之后应该是id=3才对，但结果显然并非如此。

这是因为读的时候，是使用MVCC方式，但写操作读的是已提交的记录，在终端B上，第一次读id=1，读的是mvcc的数据，执行写操作时，读到的是已提交记录id=2，再操作后就变成了id=4，而读取到最新的id=4，是因为事务中写操作之后mvcc也会发生变化。

写是通过锁控制的，写操作读取的是已提交的数据，不走MVCC的逻辑。

还有一种很有意思的情形，具体是这样的：

3.3.3.1.终端A启动事务，并insert一条记录，不提交

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from a;

+------+

| id |

+------+

| 4 |

+------+

1 row in set (0.00 sec)

mysql> insert into a(id) values(10);

Query OK, 1 row affected (22.49 sec)

mysql> select \* from a;

+------+

| id |

+------+

| 4 |

| 10 |

+------+

2 rows in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

3.3.3.2.终端B启动事务

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

1

2

3.3.3.3.终端A事务提交

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

1

2

3.3.3.4.终端B查看是只有一条记录，然后执行更新，并提交

mysql> select \* from a;

+------+

| id |

+------+

| 4 |

+------+

1 row in set (0.00 sec)

mysql> update a set id=id+100;

Query OK, 2 rows affected (4.28 sec)

Rows matched: 2 Changed: 2 Warnings: 0

mysql> select \* from a;

+------+

| id |

+------+

| 104 |

| 110 |

+------+

2 rows in set (0.00 sec)

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

有没有什么发现？

是的，对当前事务来说，插入操作是属于已提交的，即插入操作是在本事务之前的。

再来看另外一种：

3.3.4.1.终端A启动事务，并删除一条记录

mysql> select \* from a;

+------+

| id |

+------+

| 104 |

| 110 |

+------+

2 rows in set (0.00 sec)

mysql> delete from a where id=104;

Query OK, 1 row affected (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

3.3.4.2.终端B启动事务

mysql> begin;

Query OK, 0 rows affected (0.00 sec)

1

2

3.3.4.3.终端A事务提交

mysql> select \* from a;

+------+

| id |

+------+

| 110 |

+------+

1 row in set (0.00 sec)

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

1

2

3

4

5

6

7

8

9

10

3.3.4.4.终端B修改数据并提交

mysql> select \* from a;

+------+

| id |

+------+

| 104 |

| 110 |

+------+

2 rows in set (0.00 sec)

mysql> update a set id=id+10 where id=104;

Query OK, 0 rows affected (0.00 sec)

Rows matched: 0 Changed: 0 Warnings: 0

mysql> select \* from a;

+------+

| id |

+------+

| 104 |

| 110 |

+------+

2 rows in set (0.00 sec)

mysql> update a set id=id+10 where id=110;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> select \* from a;

+------+

| id |

+------+

| 104 |

| 120 |

+------+

2 rows in set (0.00 sec)

mysql> commit;

Query OK, 0 rows affected (0.00 sec)

mysql> select \* from a;

+------+

| id |

+------+

| 120 |

+------+

1 row in set (0.00 sec)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

又是一个很有意思的情况对不对？

这其实说明的是删除操作是在本事务之后提交的，或者说本事务对其它事务的删除操作是无视的，或是在本事务之后的。

3.4.serializable

串行化，最严格的事务隔离级别，最强的数据一致性保护，完全解决脏读、不可重复读、幻读的问题。

就是不支持并发，所以除了某些特殊场景，实际中极少看到。

————————————————

版权声明：本文为CSDN博主「allen.liang」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/ljl890705/article/details/106125055