Database Ex3 Report

实验目标

通过实现数据库事物编程与分布式数据库并发控制，更好的理解课堂上讲解的理论知识。

实验主要分成三部分，使用表级锁操作得到期望结果，使用行级锁操作得到期望结果， 数据库事务编程，分布式事务程序设计。

内容一：使用表级锁操作得到期望结果

1. Problem1:

   原始题目中在conn1.sql中使用了读锁，导致2无法被插入，因此在修改后的程序中使用了write锁，并在最后释放了表上的锁。

   修改后的程序如下：

```
# conn1.sql
# problem 1
drop table if exists test1;
create table if not exists test1 (
  test1_id int,
  PRIMARY KEY(test1_id)
) engine=MyISAM;
select connection_id();
insert into test1 values(1);
lock tables test1 write;              # Modify read lock to write lock
insert into test1 values(2);
unlock tables;                        # Unlock tables after we have
inserted the record
```

   运行结果为：

test1_id
1
2
3

   满足要求

2. Problem2:

   在这里我把读锁改成了local read锁，以防影响conn2.sql中的插入

之后我释放了表上的锁

```
# conn1.sql
# problem 2
drop table if exists test2;
create table if not exists test2 (
  test2_id int,
  PRIMARY KEY(test2_id)
) engine=MyISAM;
select connection_id();
insert into test2 values(1);
lock tables test2 local read;         # Modify read to local read
insert into test2 values(2);
unlock tables;
```

运行结果如下：

test2_id
1
3

满足要求

3. Problem3:

   在conn1.sql中，我只在最后释放了表上的所有锁。

   之后在conn2.sql中，我把读锁改成了写锁，因为加了读锁之后就不能加入题目中要求的test3_id = 3

   之后在最后释放了锁

```
# conn1.sql
# problem 3
drop table if exists test3;
create table if not exists test3 (
  test3_id int,
  PRIMARY KEY(test3_id)
) engine=MyISAM;
select connection_id();
insert into test3 values(1);
lock tables test3 read;
insert into test3 values(2);
unlock tables;                      # Add unlock
```

```
# conn2.sql
# problem  3
drop procedure if exists problem3;
delimiter //
create procedure problem3()
```

```
begin
  select connection_id();
  insert into test3 values(4);
  select * from test3;
end
//
delimiter ;
lock tables test3 write;          # Modify read lock to write lock
insert into test3 values(3);
unlock tables;                    # Add unlock
call problem3;
```

运行结果：

test3_id
1
3
4

满足要求

4. Problem4

   在conn1.sql中，我把local read锁改成了write锁，因为local read锁会让我们需要的记录无法被插入。并且我在最后加入了释放锁的操作。

   在conn2.sql中，我在最后释放了锁。

```
# conn1.sql
# problem 4
drop table if exists test4;
create table if not exists test4 (
  test4_id int,
  PRIMARY KEY(test4_id)
) engine=MyISAM;
select connection_id();
insert into test4 values(1);
lock tables test4 write;              # Modify local read lock to write
lock
insert into test4 values(2);
unlock tables;
```

```
# conn2.sql
# problem  4
drop procedure if exists problem4;
delimiter //
create procedure problem4()
begin
  select connection_id();
```

```
  insert into test4 values(3);
  select * from test4;
end
//
delimiter ;
lock tables test4 read;
unlock tables;                          # Add unlock
call problem4;
```

运行结果：

test4_id
1
2
3

满足要求

5. Problem5

   在这里我只在conn1.sql中最后加上了释放锁的操作

   ```
   #problem 5
   drop table if exists test5;
   create table if not exists test5 (
     test5_id int,
     PRIMARY KEY(test5_id)
   ) engine=MyISAM;
   select connection_id();
   insert into test5 values(1);
   lock tables test5 write;
   insert into test5 values(2);
   unlock tables;                                # Add unlock
   ```

   运行结果：

test5_id
1
2

   满足要求

6. Problem6

   在这里我在conn1.sql中把test6_2表上也加了写锁，mysql在一个表被锁住了另一个表没被锁住的情况下无法对另一个表进行修改，因此这里给test6_2也加入了写锁, 最后加上了释放锁的操作。

   ```
   # conn1.sql
   # problem 6
   ```

```
drop table if exists test6_1;
drop table if exists test6_2;
create table if not exists test6_1 (
  test6_1_id int,
  PRIMARY KEY(test6_1_id)
) engine=MyISAM;
create table if not exists test6_2 (
  test6_2_id int,
  PRIMARY KEY(test6_2_id)
) engine=MyISAM;
select connection_id();
insert into test6_1 values(1);
lock tables test6_1 write;
lock tables test6_2 write;
insert into test6_2 values(2);
unlock tables;
```

运行结果：

test6_2_id
2
3

满足要求

内容二：使用行级锁操作得到期望结果

1. Problem1

   由于我们设置了auto_commit为0，在conn3.sql中problem1插入的数据在没有执行commit之前并没有被插入表格中，select之后结果为空。

   因此这里做的修改就是在conn3中加入commit语句

```
# conn3.sql
# problem 1
set autocommit = 0;
drop table if exists test1;
create table if not exists test1 (
  test1_id int,
  PRIMARY KEY(test1_id)
) engine=InnoDB;
select connection_id();
insert into test1 values(1);
insert into test1 values(2);
insert into test1 values(3);
select * from test1 where test1_id = 1 for update;
COMMIT;       # Add commit
```

运行结果：

test1_id
1
NULL

满足要求

2. Problem2

首先我在conn3.sql的最后加入了COMMIT语句，把插入的数据提交到表中，并且释放了锁，因此之后在conn4.sql中的select语句结果不会为NULL。其次我在conn4.sql的最后加入了COMMIT语句，因为之前的select ... for update语句给表上加上了行级锁，通过commit语句可以释放锁。

```
# conn3.sql
# problem 2
set autocommit = 0;
drop table if exists test2;
create table if not exists test2 (
  test2_id int,
    val int,
  PRIMARY KEY(test2_id)
) engine=InnoDB;
select connection_id();
insert into test2 values(1,2);
insert into test2 values(2,3);
insert into test2 values(3,4);
select test2_id from test2 where val = 2 for update;
COMMIT;             # Add commit
```

```
# conn4.sql
# problem 2
set autocommit = 0;
drop procedure if exists problem2;
delimiter //
create procedure problem2()
begin
  select connection_id();
  select * from test2 where val = '3' for update;
  COMMIT;         # Add commit
end
//
delimiter ;
call problem2;
```

运行结果：

test2_id	val
2	3

满足要求

3. Problem3

在这里我在conn3.sql中加入了COMMIT语句，把插入的数据加入到表中，同时释放表上的锁。

```
# conn3.sql
# problem3
set autocommit = 0;
drop table if exists test3;
create table if not exists test3 (
  test3_id int,
    val int,
  PRIMARY KEY(test3_id)
) engine=InnoDB;
select connection_id();
insert into test3 values(1,2);
insert into test3 values(2,3);
insert into test3 values(3,4);
select test3_id from test3 where val = 2 lock in share mode;
COMMIT;
```

运行结果：

test3_id	val
1	100

满足要求

4. Problem4

在problem4中，我们需要之前的插入语句执行，但不希望之后的update语句执行，因此我先把之前的insert语句的结果commit，再把之后update的结果rollback回去。为了防止conn2读取到脏数据，我把共享锁改成了排他锁。

在conn4.sql中，我最后加上了COMMIT语句，释放表上的锁。

```
# conn3.sql
# problem 4
set autocommit = 0;
drop table if exists test4;
create table if not exists test4 (
  test4_id int,
    val int,
    col int,
  PRIMARY KEY(test4_id)
) engine=InnoDB;
select connection_id();
insert into test4 values(1,2,3);
insert into test4 values(2,3,4);
```

```
insert into test4 values(3,3,5);
insert into test4 values(4,5,6);
COMMIT;                                                  # Add COMMIT
select * from test4 where test4_id = 3 for Update; # Modify lock in share mode
to for update
update test4 set col= 10 where test4_id = 3;
ROLLBACK;                                                # Add Rollback
```

```
# conn4.sql
# problem 4
set autocommit = 0;
drop procedure if exists problem4;
delimiter //
create procedure problem4()
begin
  select connection_id();
  select * from test4 where test4_id = 3 lock in share mode;
  COMMIT;            # Add commit
end
//
delimiter ;
call problem4;
```

运行结果：

test4_id	val	col
3	3	5

满足要求

5. Problem5

   由于这里需要返回的值为NULL，于是我直接在创建表之后把事物commit，之后在修改表一系列操作之后把表rollback返回到刚创建表的时候。

   同样在conn4.sql中我加上了COMMIT，把加上的行级锁释放掉。

```
# conn3.sql
# problem 5
set autocommit = 0;
drop table if exists test5;
create table if not exists test5 (
  test5_id int,
    val int,
    col int,
  PRIMARY KEY(test5_id)
) engine=InnoDB;
COMMIT;                                          # Add COMMITt
select connection_id();
```

```
insert into test5 values(1,2,3);
insert into test5 values(2,3,4);
insert into test5 values(3,3,7);
insert into test5 values(4,5,6);
select * from test5 where val = 3 for update;
ROLLBACK;                                          # Add ROLLBACK
```

```
# conn4.sql
# problem 5
set autocommit = 0;
drop procedure if exists problem5;
delimiter //
create procedure problem5()
begin
  select connection_id();
    select * from test5 where col = 7 lock in share mode;
  COMMIT;
end
//
delimiter ;
call problem5;
```

运行结果：

test5_id	val	col

满足要求

6. Problem 6

在conn3.sql中，我把我们需要的插入结果先commit，之后我把不需要的修改结果直接rollback。

```
# conn3.sql
# problem 6
set autocommit = 0;
drop table if exists test6;
create table if not exists test6 (
  test6_id int,
    val int,
    col int,
  PRIMARY KEY(test6_id)
) engine=InnoDB;
select connection_id();
insert into test6 values(1,2,3);
insert into test6 values(2,3,4);
```

```
commit;                                 # Add Commit
insert into test6 values(3,3,7);
insert into test6 values(4,5,6);
ROLLBACK;                               # Add Rollback
```

运行结果：

count(*)
1

满足要求

内容三：数据库事务编程

题目要求：

场景一：

用户A：查询所有价格高于100元的商品名

用户B：将所有商品的价格设置九折优惠

场景二：

商品ID为1,2,3的剩余库存分别为50,100,200件 （初始状态）

用户A：订单中包含商品1三十件，商品2五十件

用户B：订单中包含商品2七十件，商品2八十件

Q1: 在MySQL默认的事务隔离级别下， 使用数据库事务编程接口实现上述两位用户正确的业务逻辑。

在这里我利用java的mysql接口执行相应的sql操作，具体实现见Main1.java文件。

运行结果如下所示：

1. **场景1**

为了更好的验证事务之间的执行关系，我只有一个商品（Dress）的价格大于100元，为110元，其他商品的价格小于100元。

执行前商品价格如下：

	Goods_Id	Goods_Name	Goods_Price	Goods_Stock
1	1	Biscuit	3.2	50
2	2	Juice	6.3	100
3	3	Dress	110	200

在多次运行结果之后，我的结果总是：

```
Thread2 running
Thread1 running
Goods price higher than 100: Dress
Thread1 exits
Thread2 exits
```

执行后商品价格：

	Goods_Id	Goods_Name	Goods_Price	Goods_Stock
1	1	Biscuit	2.8800000000000003	50
2	2	Juice	5.67	100
3	3	Dress	99	200

场景二：

场景二中我添加了三个商品，商品库存数量按照题目要求分别为50，100，200

	Goods_Id	Goods_Name	Goods_Price	Goods_Stock
1	1	Biscuit	3.2	50
2	2	Juice	6.3	100
3	3	Dress	110	200

执行事务后结果：

所有商品都被成功添加

	Bill_Info_Id	Bill_Id	Goods_Id	Goods_Num	Goods_Price
1	1	2	2	70	<null>
2	2	1	1	30	<null>
3	3	1	2	50	<null>
4	4	2	3	80	<null>

库存数量与正确逻辑下库存量不符合：

	Goods_Id	Goods_Name	Goods_Price	Goods_Stock
1	1	Biscuit	3.2	20
2	2	Juice	6.3	50
3	3	Dress	110	120

Q2: 分析在该隔离级别下是否存在脏读/幻读/不可重复读的问题，若有，请设计场景并验证

Answer：

在InnoDB中默认的隔离等级是Repeatable Read。在这种隔离等级中，一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，不能看到其他事务对已经提交的已有记录的更新。

mysql5.6以后幻读的问题被解决了，因此这里的查询语句在过程中不会出现变化，不会出现幻读。

由于一个事务执行期间不会出现对已有数据的更新，也不会看到没有提交的数据更新以及插入修改，因此读取数据的值不会发生改变，没有不可重复读的问题以及脏读的问题。

因此该隔离级别下不存在幻读，不可重复读以及脏读的问题。

附加题

Q1: 将事务隔离级别修改为read uncommitted，在该隔离级别下，设计脏读 /幻读/不可重复读三类场景并验证

实验测试

针对幻读以及不可重复读的问题，我设计了以下的测试。主要基于场景一。

1. **幻读**

   对于这个问题我先让用户A先执行一遍select语句，输出结果，再等到用户B的操作结束提交后在执行一遍select语句，对比两次结果

2. **不可重复读现象**

   在这里我先让用户A执行读取商品3的价格，之后让用户B执行更新操作，再之后让用户A再次读取商品3的价格，对比两者的区别

3. **脏读**

   在这里我让用户B在用户A第一次选择时候，插入了商品4（Gold，价格为200），之后rollback，观察第二次select中是否会出现gold

实验结果

```
Thread2 running
Thread1 running
First Time Select
Goods price higher than 100: Dress
Goods No.3 price: 110.0
Second Time Select
Goods price higher than 100: Gold
Goods No.3 price: 99.0
Thread1 exits
Thread2 exits
```

1. 幻读

   重复上次测试，可以看到事务用户A两次选择出的值不同，出现了幻读现象（之前不存在gold）

2. 不可重复读

   此外用户A中两次读取的值不同，出现了不可重复读的问题

3. 脏读

   并且之后在gold操作被rollback了，但是仍选取出了gold，发生了脏读问题

2. 考虑不同MySQL引擎（MyISAM和InnoDB）支持的封锁粒度，请分析出现活锁或死锁对应的隔离级别和封锁粒度，在上述在线购物系统中设计对应场景进行验证，并给出解决思路。

在MyISAM中，支持表级锁，分为读锁和写锁。而在InnoDB中支持的是行级锁，同样分成读锁和写锁。

当隔离级别为Repeatable Read的时候，可能会出现死锁。SQL Server对读取数据需要加共享锁直到事务结束才释放。然后当两个都加上共享锁的用户同时向对对方加锁的资源进行修改时，就出现死锁现象。而只有在加行级锁的情况下才能出现死锁。

在每种隔离级别中都可能出现活锁的现象。下面我设计了两个场景来分别验证活锁和死锁的想象。表级锁和行级锁都可能出现活锁现象。

活锁场景

用户A：把1号商品的价格设为100元（thread5）

用户B：把1号商品的价格设为1元(thread6)

用户C：把1号商品的价格设为3元(thread7)

用户D：把1号商品的价格置为0元(thread8)

期望结果：用户D的请求在很久不能被执行

使用mysql引擎：MyISAM（表级锁）

```
Thread5 running
Thread7 running
Thread8 running
Thread6 running
Got the lock
Got the lock
Got the lock
Release the lock
Got the lock
Thread6 exits
Thread7 exits
Release the lock
Thread8 exits
Release the lock
Thread5 exits
```

可以看到Thread5是第一个开始的，但是是最后一个结束的，发生了活锁现象，使用的是表级锁（MyISAM）。

解决思路：采用First Come First Serve的方法减轻活锁现象的影响。

死锁现象

用户A：把商品1的价格设为10， 商品2的价格设为3

用户B：把商品1的价格设为5，商品2的价格设为6

期望结果：发生死锁

mysql引擎：InnoDB

```
Thread9 running
Thread10 running
Thread 9 Got locks for Goods 1
Thread 10 Got locks for Goods 1
Thread 9 Got locks for Goods 2
Thread9 exits
com.mysql.jdbc.exceptions.jdbc4.MySQLTransactionRollbackException: Deadlock found when trying to get lock; try restarting transaction
    at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
    at java.base/java.lang.reflect.Constructor.newInstance(Constructor.java:490)
    at com.mysql.jdbc.Util.handleNewInstance(Util.java:425)
    at com.mysql.jdbc.Util.getInstance(Util.java:408)
    at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:952)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3933)
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3869)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:2524)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2675)
    at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2465)
    at com.mysql.jdbc.StatementImpl.executeUpdateInternal(StatementImpl.java:1536)
```

先用户A给商品1加上行级锁，再让B给商品2加上行级锁

之后在让用户A请求商品2的行级锁，用户B请求商品1的行级锁

最后发生以上死锁

避免死锁的方法：

1) 资源剥夺法。挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但应防止被挂起的进程长时间得不到资源，而处于资源匮乏的状态。 2) 撤销进程法。强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。撤销的原则可以按进程优先级和撤销进程代价的高低进行。

内容四：分布式事务程序设计

问题场景和作业要求：

1. 在分布式数据库场景下，用户、订单和订单详情三个关系保存在非本地数据库中，其他关系保存在本地数据库
2. 远程数据库地址：202.120.40.131
3. 用户名：学号；密码：123456；默认schema名称：db+学号。如db517021910000
4. 通过JDBC的XA事务接口成功连接数据库进行操作
5. 使用Main2.java的两阶段提交（2phasecommit）框架，实现分布式场景下，实现上述两个场景中的业务逻辑。

具体程序见Main2.java~~