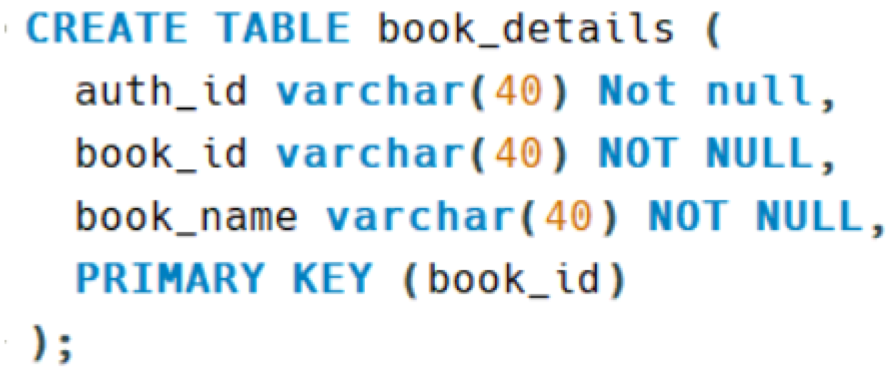
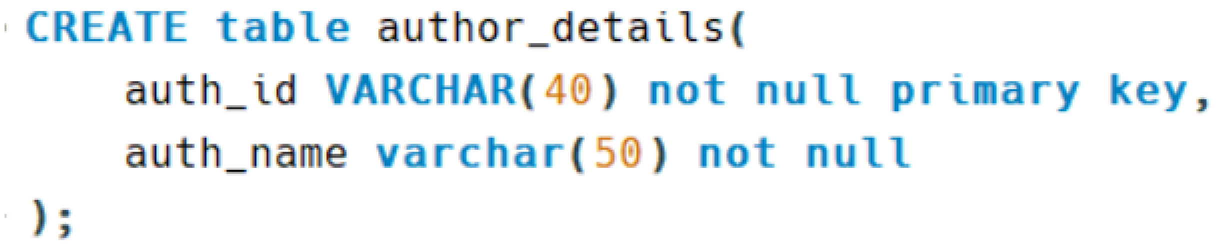
Nirbhay Sharma (B19CSE114)

1) Using the database used in lab- (7-8)

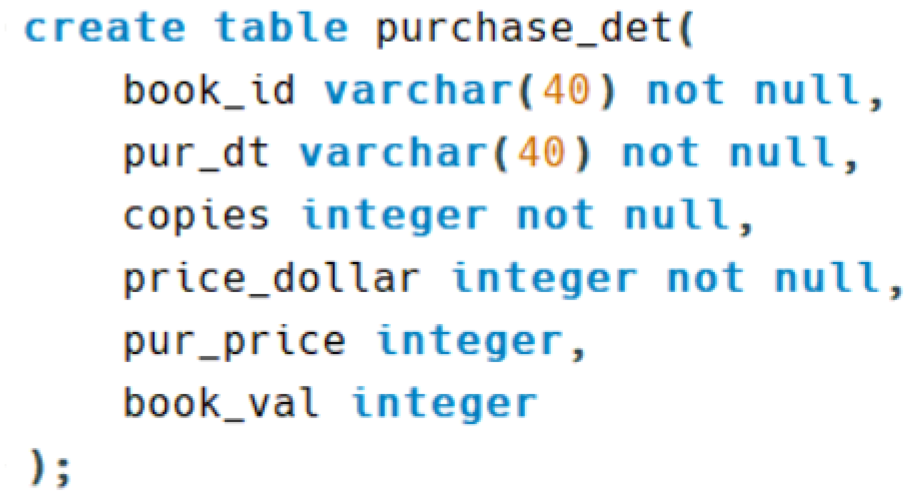
Create the required tables using following sql queries



```
CREATE TABLE book_details (
  auth_id varchar(40) Not null,
  book_id varchar(40) NOT NULL,
  book_name varchar(40) NOT NULL,
  PRIMARY KEY (book_id)
);
```



```
CREATE table author_details(
    auth_id VARCHAR(40) not null primary key,
    auth_name varchar(50) not null
);
```



```
create table purchase_det(
    book_id varchar(40) not null,
    pur_dt varchar(40) not null,
    copies integer not null,
    price_dollar integer not null,
    pur_price integer,
    book_val integer
);
```

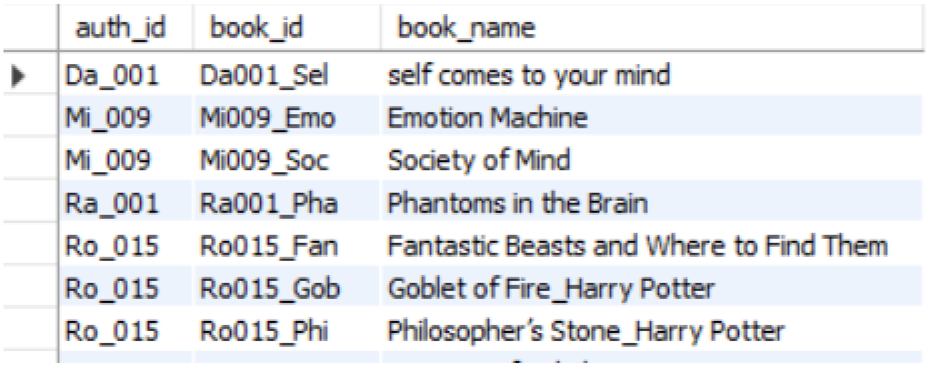
And insert the data given, to the respective tables

2) simulation of multithreaded environment

To create a multi threaded environment in sql we need to create multiple instances of sql workbench and then we need to just run different sql transaction in them, we will use the above tables and two users are simultaneously updating the variables and committing the intermediate steps are also shown below

-- user 1

```
use mytestdb;
start transaction;

SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update book_details
set book_name = "self comes to your mind"
where book_id = "Da001_Sel";
select * from book_details;
```



auth_id	book_id	book_name
Da_001	Da001_Sel	self comes to your mind
Mi_009	Mi009_Emo	Emotion Machine
Mi_009	Mi009_Soc	Society of Mind
Ra_001	Ra001_Pha	Phantoms in the Brain
Ro_015	Ro015_Fan	Fantastic Beasts and Where to Find Them
Ro_015	Ro015_Gob	Goblet of Fire_Harry Potter
Ro_015	Ro015_Phi	Philosopher's Stone_Harry Potter

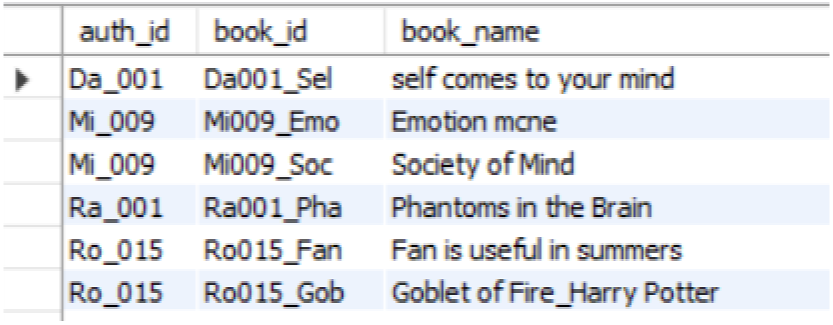
```
update book_details
set book_name = "Emotion mcne"
where book_id = "Mi009_Emo";
select * from book_details;

select * from author_details;
commit;
rollback;
```

-- user 2

```
use mytestdb;
start transaction;

SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update book_details
```

```
set book_name = "Fan is useful in summers"
where book_id = "Ro015_Fan";

select * from book_details;

update book_details
set book_name = "Emotion mcne"
where book_id = "Mi009_Emo";
select * from book_details;

select * from author_details;
commit;
rollback;
```

After both the users commit the table looks like this which means that multiple users have updated values and finally after commit it will be visible in the final database so it simulates the multithreading environment.



auth_id	book_id	book_name
Da_001	Da001_Sel	self comes to your mind
Mi_009	Mi009_Emo	Emotion mcne
Mi_009	Mi009_Soc	Society of Mind
Ra_001	Ra001_Pha	Phantoms in the Brain
Ro_015	Ro015_Fan	Fan is useful in summers
Ro_015	Ro015_Gob	Goblet of Fire_Harry Potter

4) table lock deadlock situation

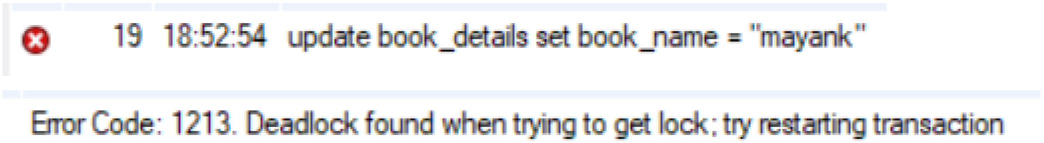
Code for user -1;

```
show databases;
set session transaction isolation level repeatable read;
-- CREATE TABLE book_details (
--   auth_id varchar(40) Not null,
--   book_id varchar(40) NOT NULL,
--   book_name varchar(40) NOT NULL,
--   PRIMARY KEY (book_id)
-- );

-- CREATE table author_details(
--     auth_id VARCHAR(40) not null primary key,
--     auth_name varchar(50) not null
-- );

-- create table purchase_det(
--     book_id varchar(40) not null,
--     pur_dt varchar(40) not null,
--     copies integer not null,
--     price_dollar integer not null,
--     pur_price integer,
--     book_val integer
-- );

SET SQL_SAFE_UPDATES=0;
set autocommit = 0;
use mytestdb;

start transaction;

update book_details set book_name = "nirbhay";
select * from book_details;

update author_details set auth_name="mayank";
select * from author_details;

commit;
rollback;
show session variables like "%transaction_isolation";
```

Code for user -2:

```
show databases;
set session transaction isolation level repeatable read;
use mytestdb;
show tables;
SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

start transaction;

update author_details set auth_name="nirbhay";
select * from author_details;

update book_details set book_name = "mayank";
```

```
select * from book_details;
commit;
rollback;
```



```
19 18:52:54 update book_details set book_name = "mayank"

Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction
```

B) in order to merge the timestamp and priority of the transaction, to decide the victim for deadlock removal we apply the following algorithm which takes the {priority, timestamp, numberofqueriessofar} so we can choose that transaction as victim for deadlock resolving which has least priority and if for two transaction priority is same then choose the one which has arrived early and if the arrival time is also same then choose the ones which has least number of queries run so far as rolling it back will take less time

C) for simulation c++ code file named deadlock_victim.cpp is attached

D) the solution should not face conflict serializability as previously it was for sure conflict serializable and we are just trying to rolling back the transaction which should not affect the conflict serializability

3) tuple lock deadlock situation

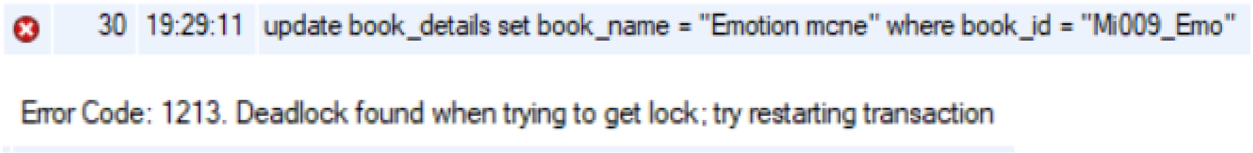
-- Code for user1;

```
SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

start transaction;

update book_details
set book_name = "self comes to your mind"
where book_id = "Da001_Sel";

update book_details
set book_name = "Emotion mcne"
where book_id = "Mi009_Emo";

select * from author_details;
commit;
rollback;
```

-- Code for user2;

```
SET SQL_SAFE_UPDATES=0;
```

```
set autocommit = 0;

start transaction;

update book_details
set book_name = "Emoticon mcney"
where book_id = "Mi009_Emo";

update book_details
set book_name = "self comes to your minds"
where book_id = "Da001_Sel";

commit;
rollback;
```

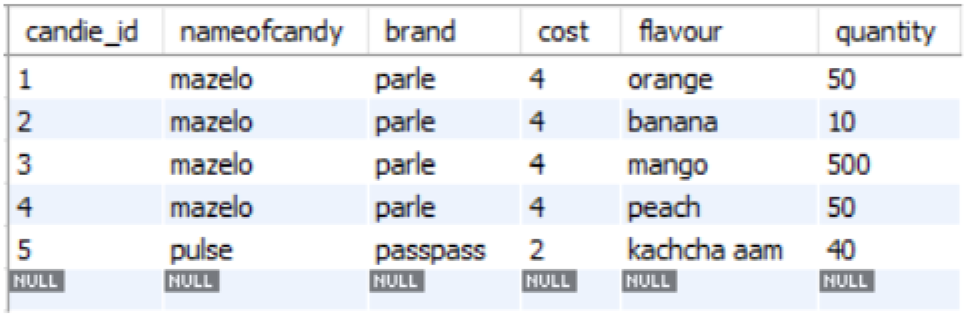


```
30 19:29:11 update book_details set book_name = "Emotion mcne" where book_id = "Mi009_Emo"

Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction
```

B) in order to merge the timestamp and priority of the transaction, to decide the victim for deadlock removal we apply the following algorithm which takes the {priority, timestamp, numberofqueriessofar} so we can choose that transaction as victim for deadlock resolving which has least priority and if for two transaction priority is same then choose the one which has arrived early and if the arrival time is also same then choose the ones which has least number of queries run so far as rolling it back will take less time
C) for simulation c++ code file named deadlock_victim.cpp is attached
D) the solution should not face conflict serializability as previously it was for sure conflict serializable and we are just trying to rolling back the transaction which should not affect the conflict serializability

5) consider a table of confectionery database

```
create database confectionery;
use confectionery;

create table candies(
        candie_id varchar(30) not null primary key,
   nameofcandy varchar(50) ,
   brand varchar(50),
   cost integer,
   flavour varchar(50),
    quantity integer
);
insert into candies values("1","mazelo","parle",4,"orange",50),
                                                    ("2","mazelo","parle",4,"banana",10),
                  ("3","mazelo","parle",4,"mango",500),
                  ("4","mazelo","parle",4,"peach",50),
                  ("5","pulse","passpass",2,"kachcha aam",40);

select * from candies;
```

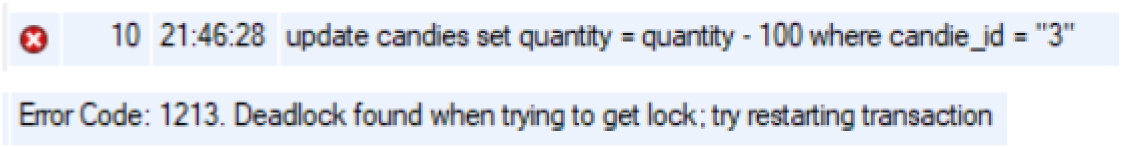


candie_id	nameofcandy	brand	cost	flavour	quantity
1	mazelo	parle	4	orange	50
2	mazelo	parle	4	banana	10
3	mazelo	parle	4	mango	500
4	mazelo	parle	4	peach	50
5	pulse	passpass	2	kachcha aam	40
NULL	NULL	NULL	NULL	NULL	NULL

```
-- for user 1
start transaction;
SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update candies
set quantity = quantity - 10
where candie_id = "2";

update candies
set quantity = quantity - 100
where candie_id = "3";

commit;
rollback;

-- for user 2
use confectionery;

start transaction;
SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update candies
set quantity = quantity - 300
where candie_id = "3";
```

update candies
set quantity = quantity - 5
where candie_id = "2";

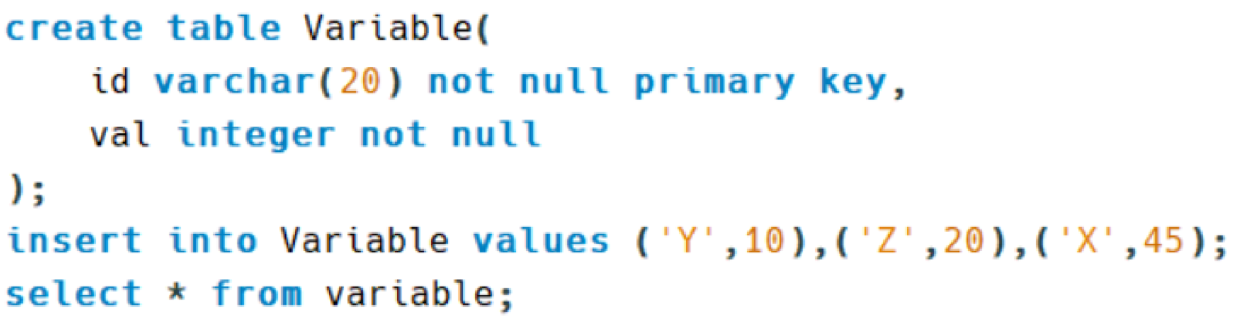
commit;
rollback;



```
10  21:46:28  update candies set quantity = quantity - 100 where candie_id = "3"
Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction
```

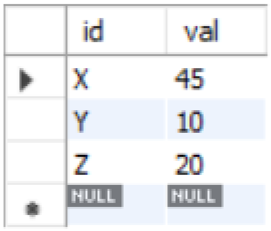
Above simple scenario is an example of deadlock occurrence where 2 users simultaneously tries to purchase candies but when user2 to purchase candies with candie id 2 , since this tuple currently acquired by user1 so user2 has to wait and similarly when user1 tries to get candies with candie id 3 then it creates a deadlock situation as two users are simultaneously waiting for each other.

6) Deadlock will occur in that scenario
Simulation using a demo database
First create the database as follows:



```
create table Variable(
    id varchar(20) not null primary key,
    val integer not null
);
insert into Variable values ('Y',10),('Z',20),('X',45);
select * from variable;
```

The table will look like the following



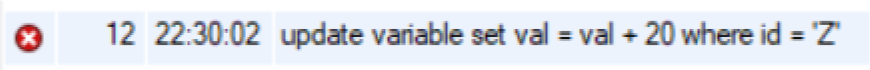
id	val
X	45
Y	10
Z	20
NULL	NULL

Now make two instances who are creating the above scenario

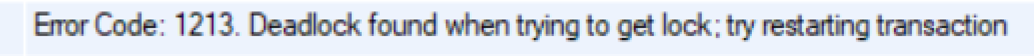
-- for user 1
use lock_data;
start transaction;

```
SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update variable
set val = val -10
where id = 'Y';

update variable
set val = val + 20
where id = 'Z';

commit;
rollback;

-- for user 2
use lock_data;

start transaction;

SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update variable
set val = val + 200
where id = 'Z';

update variable
set val = val*10
where id = 'Y';

commit;
rollback;
```

Nere we can see that user 1 is trying to access var Y and user 2 is accessing var Z but then user 2 tries to acquire var Y but since it is already occupied by user 1 it will wait for ussr 1 to release lock but then user 1 needs to acquire lock on var Z so suddenly there becomes a situation of deadlock in which user1 is waiting for user2 and vice versa.



```
12  22:30:02  update variable set val = val + 20 where id = 'Z'
```



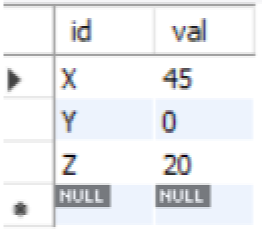
```
Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction
```

7) reconsider the above table itself

-- for user 1

```
use lock_data;
start transaction;

SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update variable
set val = val -10
where id = 'Y';
```

-- table after this operation



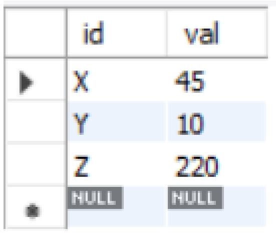
	id	val
▶	X	45
	Y	0
	Z	20
*	NULL	NULL

```
update variable
set val = val + 20
where id = 'Z';

-- deadlock occur here
commit;
rollback;
```

-- for user 2

```
use lock_data;

start transaction;

SET SQL_SAFE_UPDATES=0;
set autocommit = 0;

update variable
set val = val + 200
where id = 'Z';
```

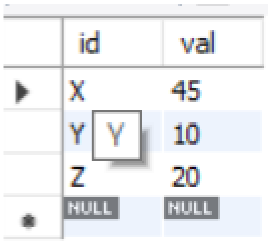
-- Table after his operation



id	val
X	45
Y	10
Z	220
NULL	NULL

```
update variable
set val = val*10
where id = 'Y';

-- It will wait here

commit;
rollback;
```

We can clearly see here the tables after each update operation from different users but once deadlock has occurred it will rollback the transaction and then it will take the system back to the safe state



id	val
X	45
Y	10
Z	20
NULL	NULL

So the table is restored after rollback

8) for simulation purposes c++ code is used which is also attached with the zip file with the name as wait_die_simulation.cpp