西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab report

Name : ABID ALI
Student_No. : 2019380141

Experiment 7

Experiment No:7

Transaction and Concurrency Control

Goal:

1. To grasp the concept transaction, and how to create the transaction.
2. Understand the data inconsistency problems in concurrency operation, and can use lock and ioslation mechanisms.

Content:

According to the requirements, try to complete the following experiments based on the database above:

1. Write a transaction to achieve the following operations: a student(student number is 20200032) transfers 200 yuan from the bank card to the campus card, and if there is a failure during the transfer process, it will be rolled back. (10 points)

Query 1

Limit to 1000 rows

```
create table icbc_card(
icbcid int,
sno varchar(8),
balance int
)ENGINE = INNODB;
create table campus_card(
sno varchar(8),
balance int
)ENGINE = INNODB;
insert into campus_card values ('20200032', 1);
insert into campus_card values ('20200033', 100);
insert into icbc_card values (1, '20200032', 300);
insert into icbc_card values (2, '20200033', 400);
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	17:07:45	create table icbc_card(icbcid int, sno varchar(8), b...	0 row(s) affected	0.031 sec
2	17:07:45	create table campus_card(sno varchar(8), balance ...	0 row(s) affected	0.032 sec
3	17:07:45	insert into campus_card values ('20200032', 1)	1 row(s) affected	0.000 sec
4	17:07:45	insert into campus_card values ('20200033', 100)	1 row(s) affected	0.000 sec
5	17:07:45	insert into icbc_card values (1, '20200032', 300)	1 row(s) affected	0.015 sec
6	17:08:01	insert into icbc_card values (2, '20200033', 400)	1 row(s) affected	0.016 sec

```
1 •  SELECT * FROM students.campus_card;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

sno	balance
20200032	1
20200033	100

Output

Action Output

#	Time	Action	Message
1	17:09:55	SELECT * FROM students.campus_card LIMIT 0, ...	2 row(s) returned

Fig: Student Campus Card

Query 1 | icbc_card

Limit to 1000 rows

```
1 •  SELECT * FROM students.icbc_card;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

icbcid	sno	balance
1	20200032	300
2	20200033	400

Result Grid

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	17:09:55	SELECT * FROM students.campus_card LIMIT 0, ...	2 row(s) returned	0.000 sec / 0.000 sec
2	17:10:36	SELECT * FROM students.icbc_card LIMIT 0, 1000	2 row(s) returned	0.015 sec / 0.000 sec

Fig: Student Icbc Card

```
use students;
SET autocommit = 1;
START TRANSACTION;
      UPDATE icbc_card
      SET balance = balance - 200
      WHERE sno = '20200032';

      update campus_card
      set balance = balance + 200
      WHERE sno = '20200032';
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	17:15:36	use students	0 row(s) affected	0.000 sec
2	17:15:36	SET autocommit = 1	0 row(s) affected	0.000 sec
3	17:15:36	START TRANSACTION	0 row(s) affected	0.000 sec
4	17:15:36	UPDATE icbc_card SET balance = balance - 2...	1 row(s) affected Rows matched: 1 Changed: 1 W...	0.000 sec
5	17:15:36	update campus_card set balance = balance + 2...	1 row(s) affected Rows matched: 1 Changed: 1 W...	0.000 sec

Fig: Updating the value

Query 1 icbc_card

Limit to 1000 rows

```
SELECT * FROM students.icbc_card;
```

Result Grid Filter Rows: Export: Wrap Cell Content:

icbcid	sno	balance
1	20200032	100
2	20200033	400

Result Grid

icbc_card 1 Read Only

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	17:16:55	SELECT * FROM students.icbc_card LIMIT 0, 1000	2 row(s) returned	0.000 sec / 0.000 sec

Fig: After transaction the updated value in icbc bank

Query 1 icbc_card campus_card

Limit to 1000 rows

```
SELECT * FROM students.campus_card;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

sno	balance
20200032	201
20200033	100

Result Grid

campus_card 1 Read Only

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	17:16:55	SELECT * FROM students.icbc_card LIMIT 0, 1000	2 row(s) returned	0.000 sec / 0.000 sec
2	17:19:41	SELECT * FROM students.campus_card LIMIT 0, ...	2 row(s) returned	0.000 sec / 0.000 sec

Fig: After transaction the updated value in campus card

Query 1

Limit to 1000 rows

```
use students;
SET autocommit = 1;
START TRANSACTION;
      UPDATE icbc_card
      SET balance = balance - 200
      WHERE sno = '20200032';

      update campus_card
      set balance = balance + 200
      WHERE sno = '20200032';

   ROLLBACK;
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	17:23:58	ROLLBACK	0 row(s) affected	0.000 sec

Fig: Doing Rollback

```
SET autocommit = 1;
START TRANSACTION;
      UPDATE icbc_card
      SET balance = balance - 200
      WHERE sno = '20200032';

      update campus_card
      set balance = balance + 200
      WHERE sno = '20200032';

   ROLLBACK;
SELECT * FROM students.campus_card;
```

Result Grid

sno	balance
20200032	1
20200033	100

campus_card 1

Output

Action Output

#	Time	Action	Message
1	17:26:37	SELECT * FROM students.campus_card LIMIT 0, 1000	2 row(s) returned

Fig: Value rolled backed to the previous value

```
SET autocommit = 1;
START TRANSACTION;
      UPDATE icbc_card
      SET balance = balance - 200
      WHERE sno = '20200032';

      update campus_card
      set balance = balance + 200
      WHERE sno = '20200032';

   ROLLBACK;
SELECT * FROM students.icbc_card;
```

Result Grid

icbcid	sno	balance
1	20200032	300
2	20200033	400

400

icbc_card 2

Output

Action Output

#	Time	Action	Message
1	17:26:37	SELECT * FROM students.campus_card LIMIT 0, 1000	2 row(s) returned
2	17:29:29	SELECT * FROM students.icbc_card LIMIT 0, 1000	2 row(s) returned

Fig: Value rolled backed to the previous value

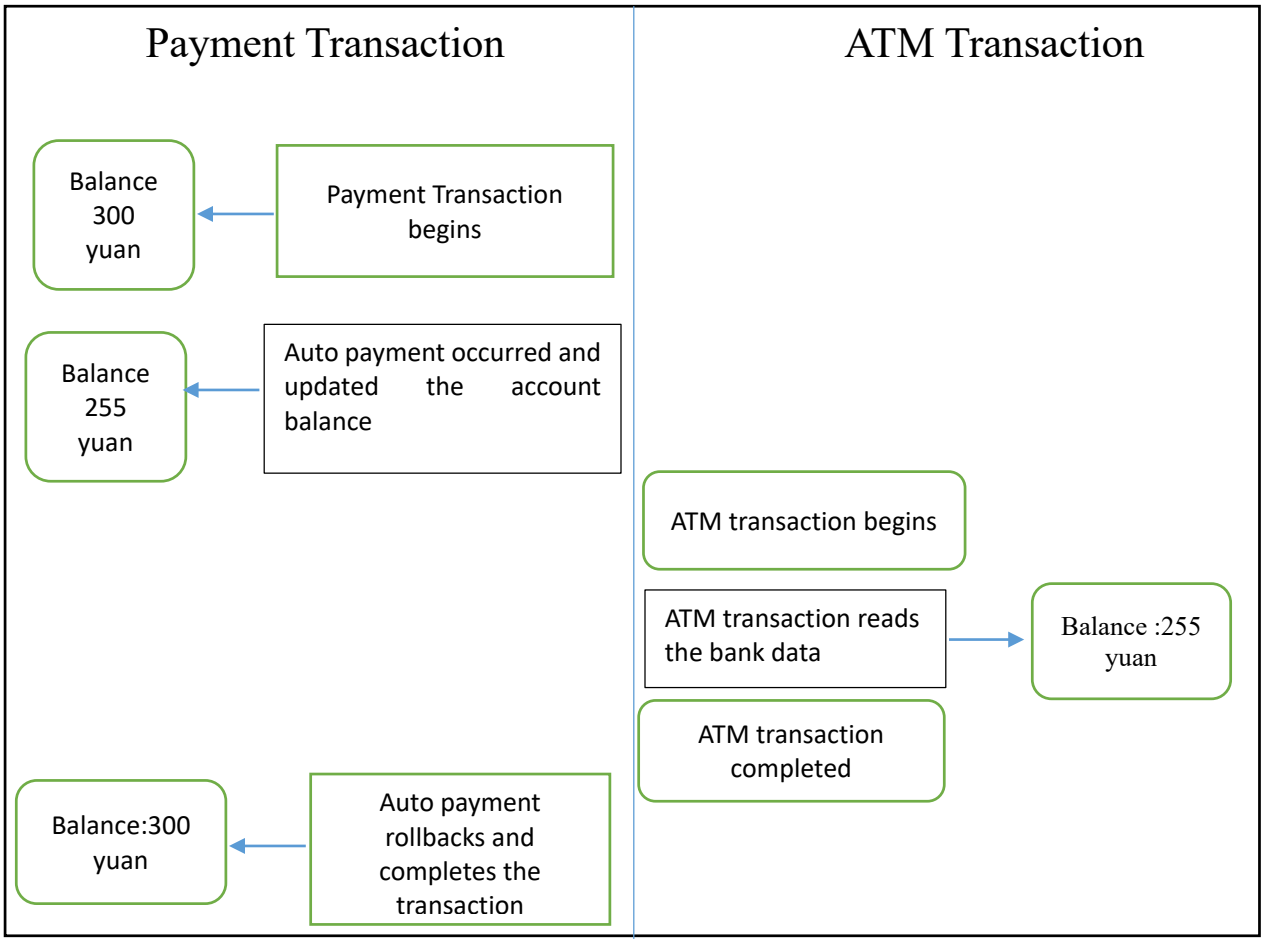
2. According to the database and tables, use specific examples to show several data inconsistency problems: such as missing and modifying, reading dirty data, non repeatable reading and phantom reading (deletion and insertion). If there is any situation that cannot be displayed, please explain the reasons. (20 points, 10 points for each data inconsistency)

Solution:

Dirty read

The simplest explanation of the dirty read is the state of reading uncommitted data. In this circumstance, we are not sure about the consistency of the data that is read because we don't know the result of the open transaction(s). After reading the uncommitted data, the open transaction can be completed with rollback. On the other hand, the open transaction can complete its actions successfully. The data that is read in this ambiguous way is defined as dirty data.

In this scenario, student number = 20200032 has 300 yuan in the bank account, and the automatic payment system withdraws 45 yuan from Betty's account for the electric bill for dorm. At that time, 20200032 wants to check the bank account on the ATM, and she notices 255 yuan in her bank account. However, if the electric bill payment transaction is rollbacked for any reason, the bank account balance will be turned to 300 yuan again, so the data read by 20200032 is dirty data. In this case, 20200032 will be confused. The following diagram illustrates this dirty read scenario in a clearer manner.



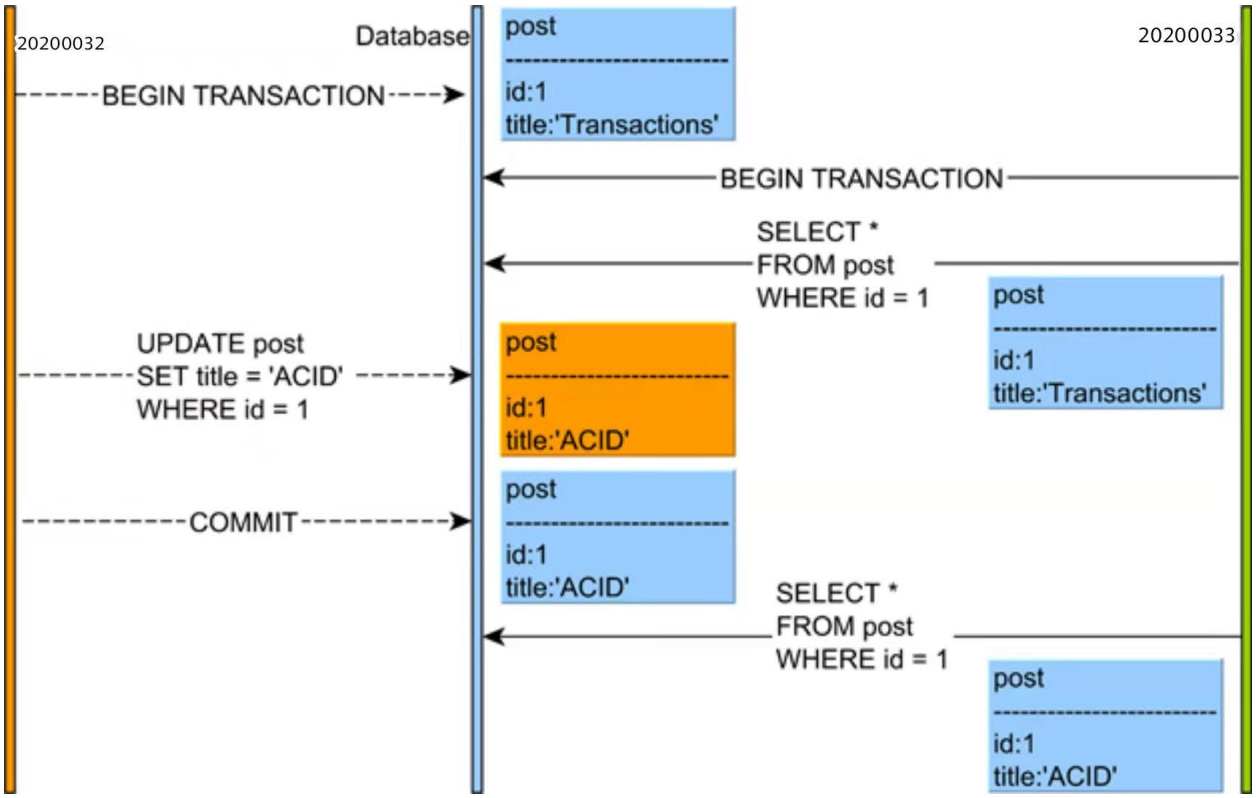
We can see that, the values are inconsistent.Therefore, we can see inconsistency.

Non-repeatable Reads:

A non-repeatable read occurs when a transaction reads the same row twice but gets different data each time. For example, suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete. If transaction 1 rereads the row, it retrieves different row values or discovers that the row has been deleted.

The **Non-Repeatable Read** anomaly looks as follows:

1. 20200032 and 20200033 start two database transactions.
2. 20200033's reads the post record and title column value is Transactions.
3. 20200032 modifies the title of a given post record to the value of ACID.
4. 20200032 commits her database transaction.
5. If 20200033's re-reads the post record, he will observe a different version of this table row

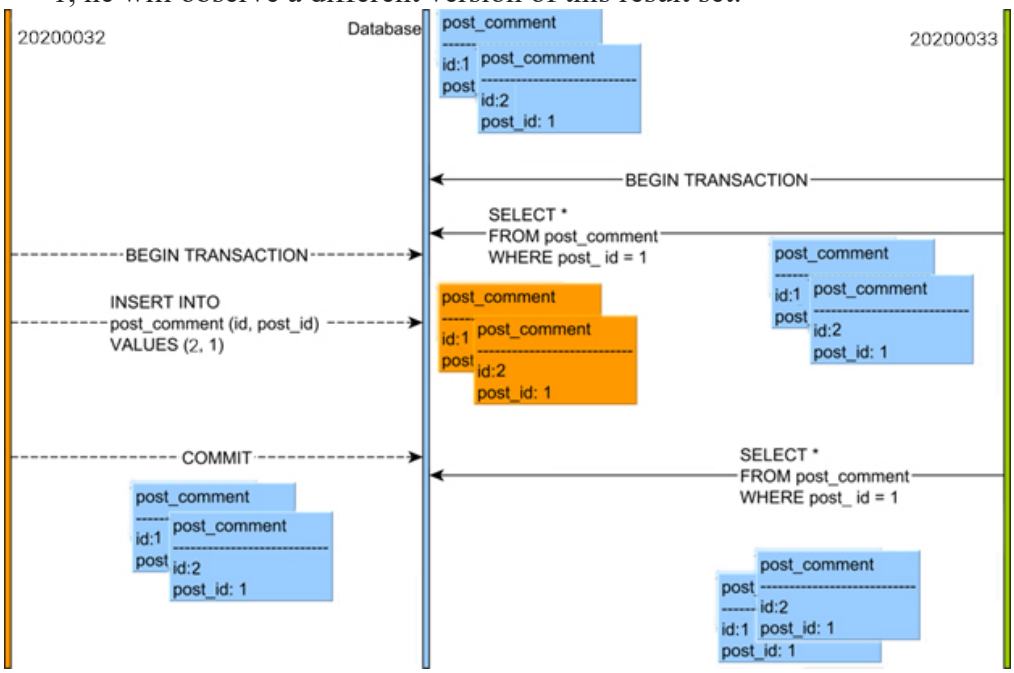


1. 20200032 and 20200033 start two database transactions.
2. 20200033's reads all the post_comment records associated with the post row with the identifier value of 1.
3. 20200032 adds a new post_comment record which is associated with the post row having the identifier value of 1.
4. 20200032 commits her database transaction.

5. If 20200033's re-reads the post_comment records having the post_id column value equal to 1, he will observe a different version of this result set.

The **Phantom Read** anomaly can happen as follows:

1. 20200032 and 20200033 start two database transactions.
2. 20200033's reads all the post_comment records associated with the post row with the identifier value of 1.
3. 20200032 adds a new post_comment record which is associated with the post row having the identifier value of 1.
4. 20200032 commits her database transaction.
5. If 20200033's re-reads the post_comment records having the post_id column value equal to 1, he will observe a different version of this result set.



So, while the **Non-Repeatable Read** applies to a single row, the **Phantom Read** is about a range of records which satisfy a given query filtering criteria.

3. By using the isolation levels or lock mechanism of the database, design solutions to solve the data inconsistency problems you have set in question 2. (20 points, 5 points for each data inconsistency)

Solution:

The SQL standard, which has been adopted by both ANSI and ISO/IEC, defines four levels of **transaction isolation**. These levels have differing degrees of impact on transaction processing throughput.

These isolation levels are defined in terms of phenomena that must be prevented between concurrently executing transactions. The preventable phenomena are:

- Dirty reads

  A transaction reads data that has been written by another transaction that has not been committed yet.

- Nonrepeatable (fuzzy) reads

  A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data. For example, a user queries a row and then later queries the same row, only to discover that the data has changed.

- Phantom reads

  A transaction reruns a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

  For example, a transaction queries the number of employees. Five minutes later it performs the same query, but now the number has increased by one because another user inserted a record for a new hire. More data satisfies the query criteria than before, but unlike in a fuzzy read the previously read data is unchanged.

Preventable Read Phenomena by Isolation Level

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Conflicting Writes in Read Committed Transactions

In a read committed transaction, a **conflicting write** occurs when the transaction attempts to change a row updated by an uncommitted concurrent transaction, sometimes called a **blocking transaction**. The read committed transaction waits for the blocking transaction to end and release its row lock. The options are as follows:

- If the blocking transaction rolls back, then the waiting transaction proceeds to change the previously locked row as if the other transaction never existed.

- If the blocking transaction commits and releases its locks, then the waiting transaction proceeds with its intended update to the newly changed row.

Read-Only Isolation Level

The **read-only isolation** level is similar to the serializable isolation level, but read-only transactions do not permit data to be modified in the transaction unless the user is SYS. Thus, read-only transactions are not susceptible to the ORA-08177 error. Read-only transactions are useful for generating reports in which the contents must be consistent with respect to the time when the transaction began.

Oracle Database achieves read consistency by reconstructing data as needed from the undo segments. Because undo segments are used in a circular fashion, the database can overwrite undo data. Long-running reports run the risk that undo data required for read consistency may have been reused by a different transaction, raising a snapshot too old error. Setting an **undo retention period**, which is the minimum amount of time that the database attempts to retain old undo data before overwriting it, appropriately avoids this problem.

REPEATABLE READ isolation level

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

to

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

and execute it. The code for second session can be used as is.

With this simple change, we have a lock that is held by first session. This leads the second session to wait for the first one to complete before actually modify row

4. Construct two transactions and update one tuple in the database at the same time. Try to use the following SQL commands to view and understand the feed back information of transaction and lock status in the current system. (10 points)

```
mysql> show engine innodb status\G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
2021-11-07 17:37:28 0x115c80 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 12 seconds
-----------------
BACKGROUND THREAD
-----------------
srv_master_thread loops: 15 srv_active, 0 srv_shutdown, 15963 srv_idle
srv_master_thread log flush and writes: 0
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 36
OS WAIT ARRAY INFO: signal count 36
RW-shared spins 0, rounds 0, OS waits 0
RW-excl spins 0, rounds 0, OS waits 0
RW-sx spins 0, rounds 0, OS waits 0
Spin rounds per wait: 0.00 RW-shared, 0.00 RW-excl, 0.00 RW-sx
------------
TRANSACTIONS
------------
Trx id counter 93590
Purge done for trx's n:o < 93589 undo n:o < 0 state: running but idle
History list length 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 284282236696736, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 284282236695960, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 284282236695184, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 284282236693632, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 284282236692856, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 284282236692080, not started
0 lock struct(s), heap size 1128, 0 row lock(s)
---TRANSACTION 93589, ACTIVE 1312 sec
4 lock struct(s), heap size 1128, 6 row lock(s), undo log entries 2
MySQL thread id 9, OS thread handle 840340, query id 631 localhost 127.0.0.1 root
Trx read view will not see trx with id >= 93590, sees < 93590
--------
FILE I/O
--------
I/O thread 0 state: wait Windows aio (insert buffer thread)
```

5. Construct a deadlock situation.

Object creation:

use student;
create table icbc_card(col1 INT,icbcid int,sno varchar(8),balance int);

insert into icbc_card values (1 ,1, '20200032', 300);
insert into icbc_card values (1,2, '20200033', 400);

create table campus_card(col1 INT,sno varchar(8),balance int);

insert into campus_card values (1,'20200032', 1);
insert into campus_card values (1,'20200033', 100);

```
SQLQuery1.sql - DE...tudent (abid (127))*
use student;
create table icbc_card(col1 INT,icbcid int,sno varchar(8),balance int);

insert into icbc_card values (1 ,1, '20200032', 300);
insert into icbc_card values (1,2, '20200033', 400);

create table campus_card(col1 INT,sno varchar(8),balance int);

insert into campus_card values (1,'20200032', 1);
insert into campus_card values (1,'20200033', 100);

100 %
Messages
(1 row affected)
(1 row affected)
Completion time: 2021-11-09T21:21:10.2494596+08:00
```

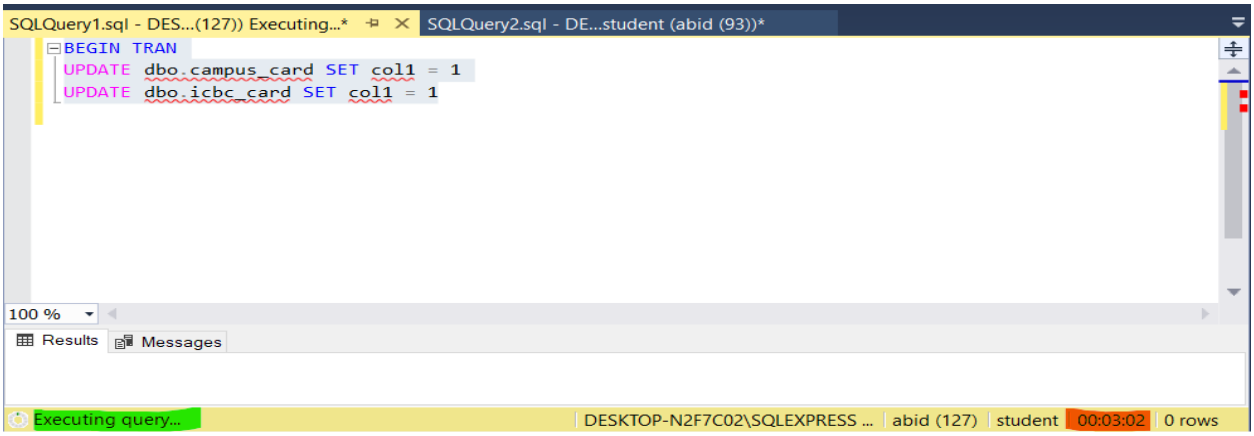
Then,we open up a new query window and paste this code and execute it

```
BEGIN TRAN
UPDATE dbo.icbc_card SET col1 = 1
```

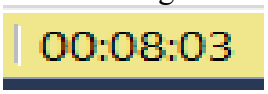
```
SQLQuery1.sql - DE...tudent (abid (127))*   SQLQuery2.sql - DE...stu
BEGIN TRAN
UPDATE dbo.icbc_card SET col1 = 1

100 %
Messages
(2 rows affected)
Completion time: 2021-11-09T21:31:21.5919331+08:00

100 %
Query executed successfully.   DESK
```

Again, we open up a new query window and paste this code and execute it

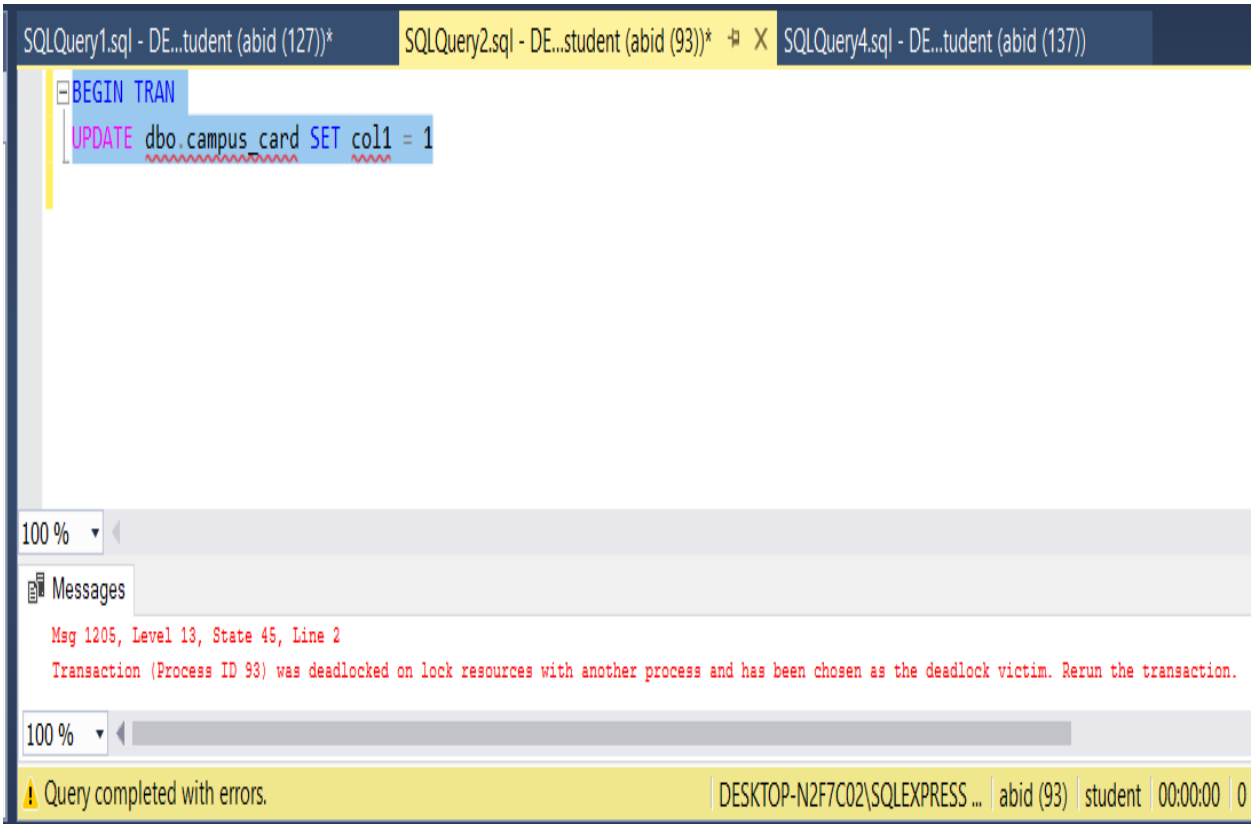
```
BEGIN TRAN
UPDATE dbo.campus_card SET col1 = 1
UPDATE dbo.icbc_card SET col1 = 1
```



After waiting certain moment of time,still the deadlock exist.



Then, we go back to your first query window (with the first BEGIN TRAN statement) and execute this code:



If we read the warning ,we can see that,we have successfully caused deadlock.

6. Construct the transaction containing some ‘savepoint’ and roll back to a savepoint at a certain time.

Solution:

Code:

```
GO
BEGIN TRANSACTION

SELECT*FROM dbo.icbc_card
insert into icbc_card values (3, '20200034', 400)
SAVE TRANSACTION SP1;


SELECT*FROM dbo.icbc_card;
insert into icbc_card values (4, '20200035', 500);
SAVE TRANSACTION SP2;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (5, '20200036', 600);
SAVE TRANSACTION SP3;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (6, '20200037', 700);
SAVE TRANSACTION SP4;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (7, '20200038', 800);
SAVE TRANSACTION S5;
```

```sql
GO
BEGIN TRANSACTION

SELECT*FROM dbo.icbc_card
insert into icbc_card values (3, '20200034', 400)
SAVE TRANSACTION SP1;


SELECT*FROM dbo.icbc_card;
insert into icbc_card values (4, '20200035', 500);
SAVE TRANSACTION SP2;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (5, '20200036', 600);
SAVE TRANSACTION SP3;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (6, '20200037', 700);
SAVE TRANSACTION SP4;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (7, '20200038', 800);
SAVE TRANSACTION S5;
```


```sql
GO
BEGIN TRANSACTION

SELECT*FROM dbo.icbc_card
insert into icbc_card values (3, '20200034', 400)
SAVE TRANSACTION SP1;


SELECT*FROM dbo.icbc_card;
insert into icbc_card values (4, '20200035', 500);
SAVE TRANSACTION SP2;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (5, '20200036', 600);
SAVE TRANSACTION SP3;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (6, '20200037', 700);
SAVE TRANSACTION SP4;

SELECT*FROM dbo.icbc_card;
insert into icbc_card values (7, '20200038', 800);
SAVE TRANSACTION S5;
```

```sql
SELECT*FROM dbo.icbc_card;
insert into icbc_card values (7, '20200038', 800);
SAVE TRANSACTION S5;
```

100 %

Results | Messages

	icbcid	sno	balance
1	1	20200032	300
2	2	20200033	400
3	3	20200034	400
4	4	20200035	500
5	5	20200036	600
6	6	20200037	700
7	7	20200038	800

Query executed successfully.

Code

ROLLBACK TO SP3;

SQLQuery1.sql - DE...tudent (abid (135))* | SQLQuery4.sql - DE...student (abid (9

```sql
ROLLBACK TRANSACTION SP3
SELECT*FROM dbo.icbc_card;
```

100 %

Messages

```
Commands completed successfully.

Completion time: 2021-11-10T10:46:01.9303152+08:00
```

```
SAVE TRANSACTION S5;


ROLLBACK TRANSACTION SP3
SELECT*FROM dbo.icbc_card;
```

100 %

Results | Messages

	icbcid	sno	balance
1	1	20200032	300
2	2	20200033	400
3	3	20200034	400
4	4	20200035	500
5	5	20200036	600

We can see that,only save point till SP3 is showing because there was 7 bank id .Now,after going to SP3 savepoint we can see 5 id.
Therefore,we have successfully created savepoints.

7. Through experiments try to check all kinds of logs in MySQL: query log, error log and slow query log.

query log/general query log

```
SET global general_log = 1;
SET global log_output = 'table';
  CREATE TABLE `general_log` (
    `event_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
                           ON UPDATE CURRENT_TIMESTAMP,
    `user_host` mediumtext NOT NULL,
    `thread_id` bigint(21) unsigned NOT NULL,
    `server_id` int(10) unsigned NOT NULL,
    `command_type` varchar(64) NOT NULL,
    `argument` mediumtext NOT NULL
  ) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='General log'
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	10:20:09	ROLLBACK TO SP3	Error Code: 1305. SAVEPOINT SP3 does not exist	0.000 sec
2	10:56:46	CREATE TABLE `general_log` (`event_time` time...	0 row(s) affected, 3 warning(s): 1681 Integer display...	0.015 sec
3	10:57:41	SET global general_log = 1	0 row(s) affected	0.000 sec
4	10:57:41	SET global log_output = 'table'	0 row(s) affected	0.000 sec
5	10:57:49	CREATE TABLE `general_log` (`event_time` time...	Error Code: 1050. Table 'general_log' already exists	0.016 sec

Query 1 icbc_card

Limit to 1000 rows

```
11    ) ENGINE=CSV DEFAULT CHARSET=utf8 COMMENT='General log'
12
13    SELECT
14      *
15  FROM
16      mysql.general_log;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

event_time	user_host	thread_id	server_id	command_type	argument
2021-11-10 10:57:49.013103	root[root] @ localhost [127.0.0.1]	9	1	Query	BLOB
2021-11-10 10:59:41.262618	root[root] @ localhost [127.0.0.1]	9	1	Query	BLOB

Result Grid

Form Editor

general_log 2 Read Only

Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	10:59:41	SELECT * FROM mysql.general_log LIMIT 0,...	2 row(s) returned	0.000 sec / 0.000 sec

slow query log

```
mysql> show global variables like '%slow%';
+-----------------------------+-------------------------+
| Variable_name               | Value                   |
+-----------------------------+-------------------------+
| log_slow_admin_statements   | OFF                     |
| log_slow_extra              | OFF                     |
| log_slow_replica_statements | OFF                     |
| log_slow_slave_statements   | OFF                     |
| slow_launch_time            | 2                       |
| slow_query_log              | ON                      |
| slow_query_log_file         | DESKTOP-N2F7C02-slow.log |
+-----------------------------+-------------------------+
7 rows in set (0.02 sec)
```

```
mysql> set global slow_query_log = 1;
```

```
    -> ^C
mysql> select @@global.long_query_time;
+---------------------------+
| @@global.long_query_time |
+---------------------------+
|                 10.000000 |
+---------------------------+
1 row in set (0.00 sec)
```

```
mysql> set global long_query_time = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> set global slow_query_log = 0;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> show create table mysql.slow_log \G
*************************** 1. row ***************************
       Table: slow_log
Create Table: CREATE TABLE `slow_log` (
  `start_time` timestamp(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6) ON UPDATE CURRENT_TIMESTAMP(6)
  `user_host` mediumtext NOT NULL,
  `query_time` time(6) NOT NULL,
  `lock_time` time(6) NOT NULL,
  `rows_sent` int NOT NULL,
  `rows_examined` int NOT NULL,
  `db` varchar(512) NOT NULL,
  `last_insert_id` int NOT NULL,
  `insert_id` int NOT NULL,
  `server_id` int unsigned NOT NULL,
  `sql_text` mediumblob NOT NULL,
  `thread_id` bigint unsigned NOT NULL
) ENGINE=CSV DEFAULT CHARSET=utf8mb3 COMMENT='Slow log'
1 row in set (0.01 sec)
```

View

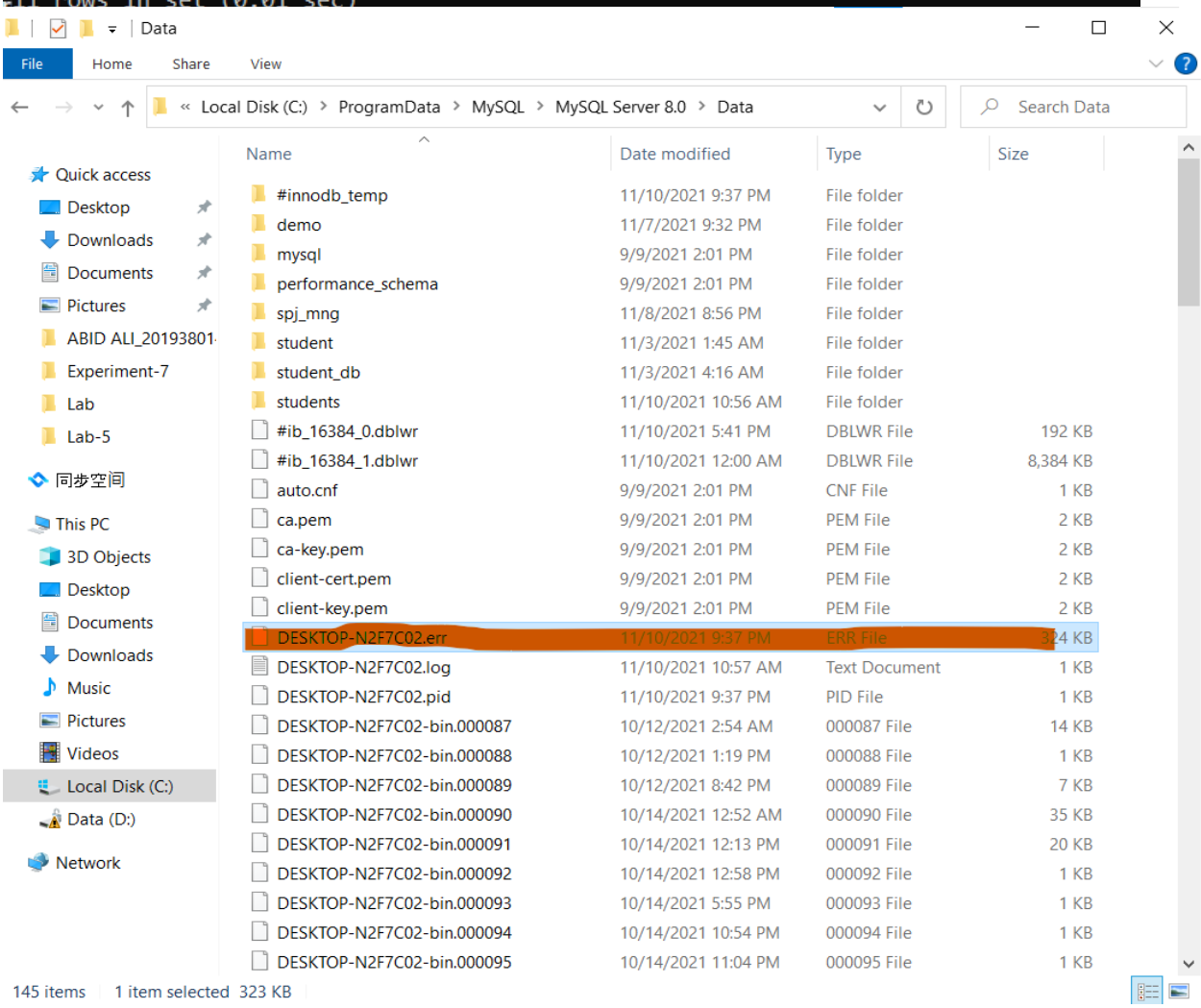
I Disk (C:) > ProgramData > MySQL > MySQL Server 8.0 > Data >

Search Data

Name	Date modified	Type	Size
DESKTOP-N2F7C02-bin.000184	11/8/2021 3:38 PM	000184 File	1 KB
DESKTOP-N2F7C02-bin.000185	11/8/2021 10:12 PM	000185 File	4 KB
DESKTOP-N2F7C02-bin.000186	11/9/2021 3:55 AM	000186 File	1 KB
DESKTOP-N2F7C02-bin.000187	11/9/2021 4:29 PM	000187 File	1 KB
DESKTOP-N2F7C02-bin.000188	11/9/2021 5:10 PM	000188 File	1 KB
DESKTOP-N2F7C02-bin.000189	11/9/2021 5:10 PM	000189 File	1 KB
DESKTOP-N2F7C02-bin.000190	11/9/2021 10:47 PM	000190 File	1 KB
DESKTOP-N2F7C02-bin.000191	11/10/2021 12:00 AM	000191 File	1 KB
DESKTOP-N2F7C02-bin.000192	11/10/2021 1:46 AM	000192 File	1 KB
DESKTOP-N2F7C02-bin.000193	11/10/2021 11:47 AM	000193 File	5 KB
DESKTOP-N2F7C02-bin.000194	11/10/2021 3:47 PM	000194 File	1 KB
DESKTOP-N2F7C02-bin.000195	11/10/2021 5:26 PM	000195 File	1 KB
DESKTOP-N2F7C02-bin.000196	11/10/2021 5:39 PM	000196 File	1 KB
DESKTOP-N2F7C02-bin.000197	11/10/2021 6:10 PM	000197 File	1 KB
DESKTOP-N2F7C02-bin.000198	11/10/2021 9:37 PM	000198 File	1 KB
DESKTOP-N2F7C02-bin.000199	11/10/2021 9:37 PM	000199 File	1 KB
DESKTOP-N2F7C02-bin.index	11/10/2021 9:37 PM	INDEX File	4 KB
DESKTOP-N2F7C02-slow.log	11/10/2021 9:37 PM	Text Document	39 KB

Error log

```
mysql> show variables like '%error%';
+-------------------------------+-----------------------------------------+
| Variable_name                 | Value                                   |
+-------------------------------+-----------------------------------------+
| binlog_error_action           | ABORT_SERVER                            |
| error_count                   | 0                                       |
| log_error                     | .\DESKTOP-N2F7C02.err                   |
| log_error_services            | log_filter_internal; log_sink_internal |
| log_error_suppression_list    |                                         |
| log_error_verbosity           | 2                                       |
| max_connect_errors            | 100                                     |
| max_error_count               | 1024                                    |
| performance_schema_error_size | 4946                                    |
| replica_skip_errors           | OFF                                     |
| slave_skip_errors             | OFF                                     |
+-------------------------------+-----------------------------------------+
11 rows in set (0.01 sec)
```



8. Use mysqlbinlog to view the transaction log of the database, and try to recover the data according to the following scenarios.

```
mysql> use db1;
Database changed
mysql> create table t1(id int);
Query OK, 0 rows affected (0.05 sec)

mysql> create table t2(id int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t1 values(11);
Query OK, 1 row affected (0.01 sec)

mysql> insert into t1 values(12);
Query OK, 1 row affected (0.00 sec)

mysql> insert into t1 values(13);
Query OK, 1 row affected (0.00 sec)

mysql> insert into t2 values(21);
Query OK, 1 row affected (0.00 sec)

mysql> insert into t2 values(22);
Query OK, 1 row affected (0.01 sec)

mysql> insert into t2 values(23);
Query OK, 1 row affected (0.00 sec)

mysql> drop table t1;
Query OK, 0 rows affected (0.02 sec)

mysql> insert into t2 values(24);
Query OK, 1 row affected (0.00 sec)

mysql>
```

We can see the binlog transaction file

```
mysql> show master status;
+---------------------------+----------+--------------+------------------+-------------------+
| File                      | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+---------------------------+----------+--------------+------------------+-------------------+
| DESKTOP-N2F7C02-bin.000201 |      156 |              |                  |                   |
+---------------------------+----------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

```
mysql>  SHOW BINLOG EVENTS IN 'DESKTOP-N2F7C02-bin.000201';
+---------------------------+------+----------------+-----------+-------------+-----------------------------------------------------------------+
| Log_name                  | Pos  | Event_type     | Server_id | End_log_pos | Info                                                            |
+---------------------------+------+----------------+-----------+-------------+-----------------------------------------------------------------+
| DESKTOP-N2F7C02-bin.000201 |    4 | Format_desc    |         1 |         125 | Server ver: 8.0.26, Binlog ver: 4                               |
| DESKTOP-N2F7C02-bin.000201 |  125 | Previous_gtids |         1 |         156 |                                                                 |
| DESKTOP-N2F7C02-bin.000201 |  156 | Anonymous_Gtid |         1 |         233 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 |  233 | Query          |         1 |         342 | use `db1`; create table t1(id int) /* xid=49 */                 |
| DESKTOP-N2F7C02-bin.000201 |  342 | Anonymous_Gtid |         1 |         419 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 |  419 | Query          |         1 |         528 | use `db1`; create table t2(id int) /* xid=50 */                 |
| DESKTOP-N2F7C02-bin.000201 |  528 | Anonymous_Gtid |         1 |         607 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 |  607 | Query          |         1 |         681 | BEGIN                                                           |
| DESKTOP-N2F7C02-bin.000201 |  681 | Table_map      |         1 |         728 | table_id: 96 (db1.t1)                                           |
| DESKTOP-N2F7C02-bin.000201 |  728 | Write_rows     |         1 |         768 | table_id: 96 flags: STMT_END_F                                  |
| DESKTOP-N2F7C02-bin.000201 |  768 | Xid            |         1 |         799 | COMMIT /* xid=51 */                                             |
| DESKTOP-N2F7C02-bin.000201 |  799 | Anonymous_Gtid |         1 |         878 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 |  878 | Query          |         1 |         952 | BEGIN                                                           |
| DESKTOP-N2F7C02-bin.000201 |  952 | Table_map      |         1 |         999 | table_id: 96 (db1.t1)                                           |
| DESKTOP-N2F7C02-bin.000201 |  999 | Write_rows     |         1 |        1039 | table_id: 96 flags: STMT_END_F                                  |
| DESKTOP-N2F7C02-bin.000201 | 1039 | Xid            |         1 |        1070 | COMMIT /* xid=52 */                                             |
| DESKTOP-N2F7C02-bin.000201 | 1070 | Anonymous_Gtid |         1 |        1149 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 | 1149 | Query          |         1 |        1223 | BEGIN                                                           |
| DESKTOP-N2F7C02-bin.000201 | 1223 | Table_map      |         1 |        1270 | table_id: 96 (db1.t1)                                           |
| DESKTOP-N2F7C02-bin.000201 | 1270 | Write_rows     |         1 |        1310 | table_id: 96 flags: STMT_END_F                                  |
| DESKTOP-N2F7C02-bin.000201 | 1310 | Xid            |         1 |        1341 | COMMIT /* xid=53 */                                             |
| DESKTOP-N2F7C02-bin.000201 | 1341 | Anonymous_Gtid |         1 |        1420 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 | 1420 | Query          |         1 |        1494 | BEGIN                                                           |
| DESKTOP-N2F7C02-bin.000201 | 1494 | Table_map      |         1 |        1541 | table_id: 97 (db1.t2)                                           |
| DESKTOP-N2F7C02-bin.000201 | 1541 | Write_rows     |         1 |        1581 | table_id: 97 flags: STMT_END_F                                  |
| DESKTOP-N2F7C02-bin.000201 | 1581 | Xid            |         1 |        1612 | COMMIT /* xid=54 */                                             |
| DESKTOP-N2F7C02-bin.000201 | 1612 | Anonymous_Gtid |         1 |        1691 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 | 1691 | Query          |         1 |        1765 | BEGIN                                                           |
| DESKTOP-N2F7C02-bin.000201 | 1765 | Table_map      |         1 |        1812 | table_id: 97 (db1.t2)                                           |
| DESKTOP-N2F7C02-bin.000201 | 1812 | Write_rows     |         1 |        1852 | table_id: 97 flags: STMT_END_F                                  |
| DESKTOP-N2F7C02-bin.000201 | 1852 | Xid            |         1 |        1883 | COMMIT /* xid=55 */                                             |
| DESKTOP-N2F7C02-bin.000201 | 1883 | Anonymous_Gtid |         1 |        1962 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 | 1962 | Query          |         1 |        2036 | BEGIN                                                           |
| DESKTOP-N2F7C02-bin.000201 | 2036 | Table_map      |         1 |        2083 | table_id: 97 (db1.t2)                                           |
| DESKTOP-N2F7C02-bin.000201 | 2083 | Write_rows     |         1 |        2123 | table_id: 97 flags: STMT_END_F                                  |
| DESKTOP-N2F7C02-bin.000201 | 2123 | Xid            |         1 |        2154 | COMMIT /* xid=53 */                                             |
| DESKTOP-N2F7C02-bin.000201 | 2154 | Anonymous_Gtid |         1 |        2231 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 | 2231 | Query          |         1 |        2356 | use `db1`; DROP TABLE `t1` /* generated by server */ /* xid=57 */ |
| DESKTOP-N2F7C02-bin.000201 | 2356 | Anonymous_Gtid |         1 |        2435 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'                            |
| DESKTOP-N2F7C02-bin.000201 | 2435 | Query          |         1 |        2509 | BEGIN                                                           |
| DESKTOP-N2F7C02-bin.000201 | 2509 | Table_map      |         1 |        2556 | table_id: 97 (db1.t2)                                           |
| DESKTOP-N2F7C02-bin.000201 | 2556 | Write_rows     |         1 |        2596 | table_id: 97 flags: STMT_END_F                                  |
| DESKTOP-N2F7C02-bin.000201 | 2596 | Xid            |         1 |        2627 | COMMIT /* xid=58 */                                             |
```

Delete

We can see the operation that was performed.

```
mysql> mysqlbinlog.exe DESKTOP-N2F7C02-bin.000201 > test_000201;
```

We restored the information in text file

```
/*!0098 set@@session.immediate_server_version =6090*//*!*/;
table_id: 97 (db1.t2)
table_id: 97 flags: STMT_END_F
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'
use `db1`; DROP TABLE `t1` /* generated by server */ /* xid=57 */
/*!*/;
# at 2356
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'
#original_commit_timestamp 22:22:43 server id 1 end_log_pos 4433 CRJ54 0xed43
#immediatel_commit_timestamp 22:22:43 server id 1 end_log_pos 4433 CRJ54 0xed43
table_id: 97 (db1.t2)
#original_commit_timestamp 22:22:43 server id 1 end_log_pos 4499 CRJ54 0xed43
#immediate_commit_timestamp 22:22:43 server id 1 end_log_pos 4499 CRJ54 0xed43
table_id: 97 flags: STMT_END_F
COMMIT /* xid=58 */
```

We can see the delete operation that we used in the mysql query and restore our data from there.

Problems:
At first ,I was having problem with the concept transaction, and how to create the transaction. Didn't know how to implement data inconsistency problems in concurrency operation, and can use lock and ioslation mechanisms.These topics were mixed of theoretical and practical concept was struggling at the beginning.

Solutions:

To solve these problems which I faced during doing this practical, I took help from internet especially YouTube,StackOverflow and W3school to get information about these errors for the solution. I also asked the teacher to help me understand them. And provided instructions helped to solve some of my errors during the experiment.

Attachments:
1) DB7_2019380141_ABID ALI.pdf

References:

1) https://www.w3schools.com/

2) https://stackoverflow.com/

3) https://youtube.com/

4) https://www.thegeekstuff.com/2017/08/mysqlbinlog-examples/

5) https://www.youtube.com/watch?v=iCizaSoJd5w

6) https://www.youtube.com/watch?v=xYysvuDAX70