Last updated: Jun 14, 2019

[notes] MariaDB Transaction Isolation Levels Sample

1. Definition

1.1. Transaction, 事务

"An SQL-transaction (transaction) is a sequence of executions of SQL-statements that is atomic with respect to recovery. That is to say: either the execution result is completely successful, or it has no effect on any SQL-schemas or SQL-data."
— The SQL Standard
The InnoDB storage engine supports ACID-compliant transactions.

[1] Transactions, https://mariadb.com/kb/en/library/transactions/

1.2. ACID, 事务的四个特性

Atomic（原子性）	所有语句作为一个单元全部成功执行或全部取消。	
Consistent（一致性）	如果数据库在事务开始时处于一致状态，则在执行该事务期间将保留一致状态。	A transaction must preserve database consistency - if a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction.
Isolated（隔离性）	事务之间不相互影响。 InnoDB的隔离性通过undo(恢复机制)和redo来实现。	为了避免事务级联回滚的情况发生
Durable（持久性）	事务成功完成后，所做的所有更改都会准确地记录在数据库中。所做的更改不会丢失。	

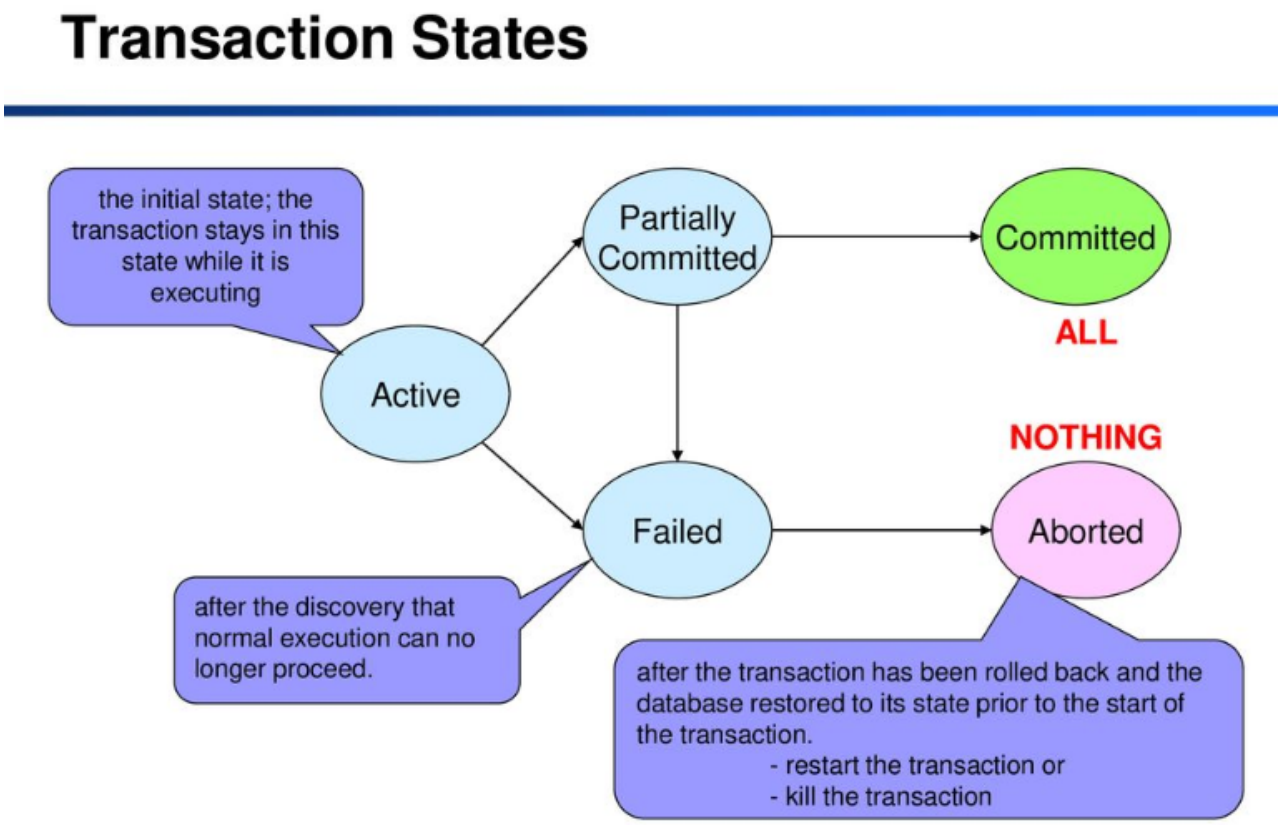
[2] ACID, https://en.wikipedia.org/wiki/ACID_(computer_science)
[3] ACID: Concurrency Control with Transactions, https://mariadb.com/kb/en/library/acid-concurrency-control-with-transactions/
[4] Disambiguating ACID and CAP, https://www.voltdb.com/blog/2015/10/22/disambiguating-acid-cap/
[5] InnoDB and the ACID Model, https://dev.mysql.com/doc/refman/5.7/en/mysql-acid.html

1.3. Transaction States, 事务的状态

ACTIVE, Committed & Failed

	事务的初始状态，表示事务正在执行
Partially Committed	after the final statement has been executed. 在最后一条语句执行之后
Failed	after the discovery that normal execution can no longer proceed. 发现事务无法正常执行之后
Aborted	after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted: 1. restart the transaction. can be done only if no internal logical error 2. kill the transaction 事务被回滚并且数据库恢复到了事务进行之前的状态之后
Committed	after successful completion 成功执行整个事务

事务状态图示



[6] Transaction Concept, https://slideplayer.com/slide/7907882/
[7] Transaction States, https://slideplayer.com/slide/14814031/
[8] Database System Concepts 第 14 章

Dirty Read	脏读	When a transaction reads the changes made by **another uncommitted** transaction 一个事务读取另一个未提交的事务所做的更改	>> Test Case 1 >> Test Case 2
Non-Repeatable Read	不可重复读	When changes from **another committed** transaction cause a prior read operation to be non-repeatable 另一个已提交的事务所做的更改导致先前读取操作不可重复	>> Test Case 3 >> Test Case 4
Phantom Read	幻读	A row that appears but was not previously visible within the same transaction 某行以前未在同一事务中显示，而现在显示出来	>> Test Case 5 >> Test Case 6

[9] Isolation, https://en.wikipedia.org/wiki/Isolation_(database_systems)

1.5. 事务的隔离级别

SQL标准中的四种隔离级别

Isolation Levels	Define	Problem	Dirty Read	Non-Repeatable Read	Phantom Read
READ UNCOMMITTED	Allows a transaction to see **uncommitted** changes made by other transactions 允许事务查看其他事务所进行的未提交更改	可能发生脏读、不可重复读和幻读问题	√	√	√
READ COMMITTED	Allows a transaction to see **committed** changes made by other transactions 允许事务查看其他事务所进行的已提交更改	可能发生不可重复读和幻读问题，但是不可以发生脏读问题 -- 只对记录加记录锁，而不会在记录之间加间隙锁，所以允许新的记录插入到被锁定记录的附近，所以在多次使用查询语句时，可能得到不同的结果	×	√	√
REPEATABLE	Ensures **consistent**	可能发生幻读问题，	×	×	√

	输出一致 InnoDB 的默认级别	数据会返回第一次查询的快照，不会返回不同的数据行			于“可重复读”将使用快照
SERIALIZABLE	Completely isolates the effects of a transaction from others 将一个事务的结果与其他事务完全隔离	各种问题都不可以发生	×	×	×

MySQL中支持的四种隔离级别

InnoDB offers all four transaction isolation levels described by the **SQL:1992 standard**: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The default isolation level for InnoDB is REPEATABLE READ.

对于MySQL/InnoDB，在REPEATABLE READ隔离级别下，可以通过加锁（**next-key locks**）策略来保证避免幻读。
REPEATABLE READ uses non-locking read views
READ COMMITTED is similar to REPEATABLE READ, but the read view is Created at the start of each statement, on the first read of an InnoDB record

[10] MySQL 四种事务隔离级的说明, https://www.cnblogs.com/zhoujinyi/p/3437475.html
[11] Deep Dive: InnoDB Transactions and Write Paths, https://mariadb.org/wp-content/uploads/2018/02/Deep-Dive_-InnoDB-Transactions-and-Write-Paths.pdf
[12] Transaction Isolation Levels, https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html

1.6. Engines that support transaction, 支持事务的存储引擎

InnoDB, SEQUENCE, NDB(MySQL), XtraDB(Percona), TokuDB(Percona)

```
select version();
+---------------------+
| version()           |
+---------------------+
| 10.3.14-MariaDB-log |
+---------------------+
1 row in set (0.00 sec)

show engines;
+--------------------+---------+------------------------------------------------------------
----------------------------+--------------+------+------------+
```

```
----------------------------+-------------+------+------------+
| MRG_MyISAM         | YES     | Collection of identical MyISAM
tables                                          | NO           | NO   |
NO         |
| CSV                | YES     | Stores tables as CSV
files                                                 | NO           | NO   |
NO         |
| MEMORY             | YES     | Hash based, stored in memory, useful for temporary
tables                        | NO           | NO   | NO         |
| MyISAM             | YES     | Non-transactional engine with good performance and
small data footprint          | NO           | NO   | NO         |
| CONNECT            | YES     | Management of External Data (SQL/NOSQL/MED),
including many file formats         | NO           | NO   | NO         |
| Aria               | YES     | Crash-safe tables with MyISAM
heritage                                        | NO           | NO   |
NO         |
| InnoDB             | DEFAULT | Supports transactions, row-level locking, foreign
keys and encryption for tables | YES          | YES  | YES        |
| PERFORMANCE_SCHEMA | YES     | Performance
Schema                                                          |
NO           | NO   | NO         |
| SEQUENCE           | YES     | Generated tables filled with sequential
values                                    | YES          | NO   | YES        |
+--------------------+---------+-----------------------------------------------------
----------------------------+-------------+------+------------+
9 rows in set (0.00 sec)
```

1.7. 事务的两阶段提交及双1问题

- innodb_flush_log_at_trx_commit=1
- sync_binlog=1

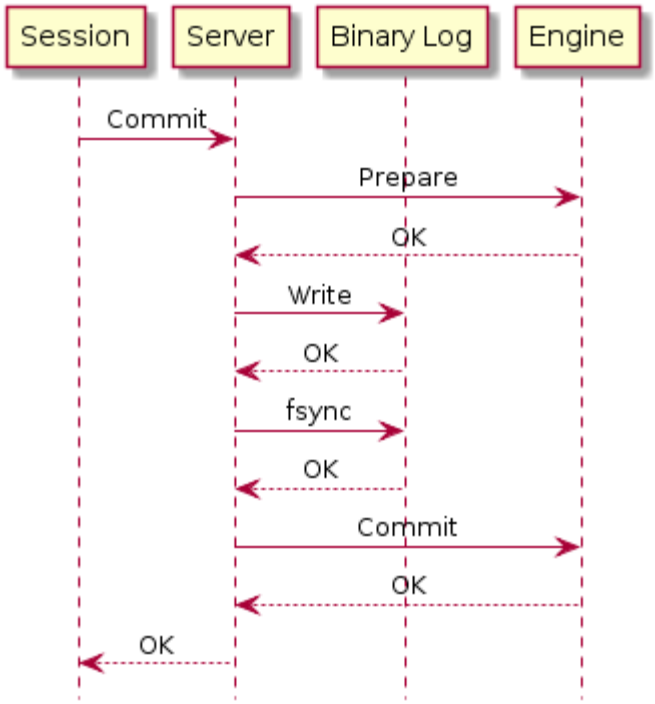
innodb_flush_log_at_trx_commit 保证了事务在InnoDB存储引擎内的修改持久化到了磁盘（redo log 持久化）
sync_binlog 保证了该事务在binlog中的修改持久化到了磁盘

```
@startuml
hide footbox
skinparam sequenceMessageAlign center

participant Session as A
participant Server as B
participant "Binary Log" as C
participant Engine as D

A -> B : Commit
B -> D : Prepare
D --> B : OK
```

```
B -> D : Commit
D --> B : OK
B --> A : OK
@enduml
```



1.8. Others

相关的表：INFORMATION_SCHEMA
innodb_trx, innodb_locks和innodb_lock_waits

```
show engine innodb status\G
...
------------
TRANSACTIONS
------------
Trx id counter 83081
Purge done for trx's n:o < 83080 undo n:o < 0 state: running but idle
History list length 15  -- 代表undo log的数量，未被清理的已提交事务的撤销日志
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 421182369247496, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
---TRANSACTION 421182369243280, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
---TRANSACTION 83080, ACTIVE 4667 sec
1 lock struct(s), heap size 1136, 0 row lock(s), undo log entries 1
MySQL thread id 10, OS thread handle 139707391616768, query id 42 localhost root
Trx read view will not see trx with id >= 83080, sees < 83080
…
```

```
tx_read_only

innodb_doublewrite
innodb_flush_log_at_trx_commit

innodb_lock_wait_timeout
innodb_rollback_on_timeout

wsrep_OSU_method
wsrep_sync_wait
```

tx_read_only
https://mariadb.com/kb/en/library/server-system-variables/#tx_read_only
Description: Default transaction access mode. If set to OFF, the default, access is read/write. If set to ON, access is read-only. The SET TRANSACTION statement can also change the value of this variable. See SET TRANSACTION and START TRANSACTION.
Commandline: --transaction-read-only=#
Scope: Global, Session
Dynamic: Yes
Type: boolean
Default Value: OFF-READ
Introduced: MariaDB 10.0

s1> insert tbl select 100,null,null,null;
ERROR 1792 (25006): Cannot execute statement in a **READ ONLY** transaction
-->
强约束事务为只读模式，相对于读写事务开销更小。

2. Sample Code

实验前提：使用MariaDB/InnoDB，且关闭自动提交。

```
Engine=InnoDB
SET autocommit = OFF;
```

No	Isolation Level	Problem	Desc
Test Case 1	READ UNCOMMITTED	Dirty Read	假定事务 **T1 修改了某行**。如果事务 **T2 读取**该行，并**发现修改**内容，但T1 尚未提交，则会出现"脏"读问题。之所以会成为一个问题，是因为如果 T1 回滚，所做的更改会被撤消，但 T2 并不会意识到这一点。

Test Case 3	READ COMMITTED	Non-Repeatable Read	如果稍后在同一个事务中**重复相同的读取操作**，则会**产生不同的结果集**。
Test Case 4	REPEATABLE READ	Non-Repeatable Read	将隔离级别升级到 REPEATABLE COMMITTED，不会出现case 3中出现的"不可重复读"问题。
Test Case 5	REPEATABLE READ	Phantom Read	如果发生幻读：假定事务 T1 和 T2 开始，并且 T1 读取了某些行。如果 T2 插入一个新行，而 T1 在重复相同的读取操作时发现该行，则会发生虚读问题（新行会成为虚行）。 但是：由于InnoDB的MVCC功能，会记录下T1中第一次查询的快照，故此问题不会发生。
Test Case 6	SERIALIZABLE	Phantom Read	序列化会等待被锁定的行，并且总是读取最新提交的数据。
Test Case 7	REPEATABLE READ	Consistent Nonlocking Reads	

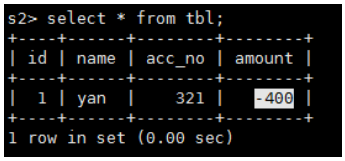
```
create table if not exists tbl(id int primary key, name varchar(20), acc_no int,
amount int);
select @@global.tx_isolation, @@session.tx_isolation;
```

Case 1: Dirty Read, READ UNCOMMITTED

在"READ UNCOMMITTED"级别下，进行"Dirty Read"测试

session 1	**session 2**	**memo**
init		
set global tx_isolation = 'READ-UNCOMMITTED'; set global autocommit=0; drop table if exists tbl; create table if not exists tbl(id int primary key, name varchar(20),		set isolation level off auto commit create test table init table data

mysql -uroot sbtest --prompt='s1> '	mysql -uroot sbtest --prompt='s2> '	login
select @@global.tx_isolation, @@session.tx_isolation\G	select @@global.tx_isolation, @@session.tx_isolation\G	check isolation level -- @@global.tx_isolation: READ-UNCOMMITTED @@session.tx_isolation: READ-UNCOMMITTED
select @@global.autocommit, @@session.autocommit\G	select @@global.autocommit, @@session.autocommit\G	check auto commit -- @@global.autocommit: 0 @@session.autocommit: 0
test		
begin;	begin;	初始状态查询结果一样 -- select * from tbl; s1> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec) s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec)
update tbl set amount=amount+200 where acc_no=321; -- 资金转入200		账户中增加200
	select * from tbl;	在session 2中账户金额变为300 s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 300 | +----+------+--------+--------+ 1 row in set (0.00 sec) --> 此时，出现"Dirty Read"。
rollback; -- 事务1执行回滚操作		此时，如果在session 1,2中做查询，结果都是100
	update tbl set amount=amount-	但由于在session 2中之前查

	500 where acc_no=321;	询的结果为300，也不知道 值的异常。
	commit; -- 事务2执行成功	

Case 2: Dirty Read, READ COMMITTED

隔离级别升级到"READ COMMITTED"，并进行与Case1相同流程的测试，则会得到不同的测试结果。
当事务1数据变更后，事务2读取该行数据，并不会发现数据发生变化。

session 1	session 2	memo
init		
set global tx_isolation = **'READ-COMMITTED'**; set global autocommit=0; drop table if exists tbl; create table if not exists tbl(id int primary key, name varchar(20), acc_no int, amount int); insert tbl select 1,'yan',321,100; commit;		set isolation level off auto commit create test table init table data
mysql -uroot sbtest --prompt='s1> '	mysql -uroot sbtest --prompt='s2> '	login
select @@global.tx_isolation, @@session.tx_isolation\G	select @@global.tx_isolation, @@session.tx_isolation\G	check isolation level -- @@global.tx_isolation: READ-COMMITTED @@session.tx_isolation: READ-COMMITTED
select @@global.autocommit, @@session.autocommit\G	select @@global.autocommit, @@session.autocommit\G	check auto commit -- @@global.autocommit: 0 @@session.autocommit: 0
test		
begin;	begin;	初始状态查询结果一样 -- select * from tbl;

		+----+------+--------+------+ | 1 | yan | 321 | 100 | +----+------+--------+------+ 1 row in set (0.00 sec)
update tbl set amount=amount+200 where acc_no=321; -- 资金转入200		账户中增加200
	select * from tbl;	在session 2中账户金额不变，并不会因为事务1的改变而发生变化 s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec) --> 此时，并未出现"Dirty Read"。
rollback;		执行回滚操作，结束测试。

Case 3: Non-Repeatable Read, READ COMMITTED

在隔离级别"READ COMMITTED"下，进行不可重复读测试。
在事务1中对数据进行变更，在事务1提交前后，分别在事务2中查询该行数据，得到的结果不同。

session 1	session 2	memo
init		
set global tx_isolation = **'READ-COMMITTED'**; set global autocommit=0; drop table if exists tbl; create table if not exists tbl(id int primary key, name varchar(20), acc_no int, amount int); insert tbl select 1,'yan',321,100; commit;		set isolation level off auto commit create test table init table data

select @@global.tx_isolation, @@session.tx_isolation\G	select @@global.tx_isolation, @@session.tx_isolation\G	check isolation level -- @@global.tx_isolation: READ-COMMITTED @@session.tx_isolation: READ-COMMITTED
select @@global.autocommit, @@session.autocommit\G	select @@global.autocommit, @@session.autocommit\G	check auto commit -- @@global.autocommit: 0 @@session.autocommit: 0
test		
begin;	begin;	初始状态查询结果一样 -- select * from tbl; +----+------+--------+------+ | id	name	acc_no	amount | +----+------+--------+------+ | 1	yan	321	100 | +----+------+--------+------+ 1 row in set (0.00 sec)
update tbl set amount=amount+200 where acc_no=321; -- 资金转入200		账户中增加200
	select * from tbl;	在session 2中查询账户金额，金额未发生变化 s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec)
commit; -- 提交事务，确认向该账户中转入200		
	select * from tbl;	待事务1提交之后，再次在事务2中发出查询，但是发现两次查询的结果**不同**。 s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount |

		此时，遇到“Non-Repeatable Read”问题。

Case 4: Non-Repeatable Read, REPEATABLE READ

升级隔离级别到”REPEATABLE READ”，并再次进行不可重复读测试。
在事务1中对数据进行变更，在事务1提交前后，分别在事务2中查询该行数据，应该得到相同的结果。

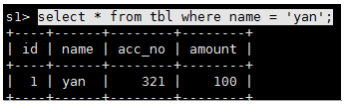
session 1	session 2	memo
init		
set global tx_isolation = **'REPEATABLE-READ'**; set global autocommit=0; drop table if exists tbl; create table if not exists tbl(id int primary key, name varchar(20), acc_no int, amount int); insert tbl select 1,'yan',321,100; commit;		set isolation level off auto commit create test table init table data
mysql -uroot sbtest --prompt='s1> '	mysql -uroot sbtest --prompt='s2> '	login
select @@global.tx_isolation, @@session.tx_isolation\G	select @@global.tx_isolation, @@session.tx_isolation\G	check isolation level -- @@global.tx_isolation: **REPEATABLE-READ** @@session.tx_isolation: **REPEATABLE-READ**
select @@global.autocommit, @@session.autocommit\G	select @@global.autocommit, @@session.autocommit\G	check auto commit -- @@global.autocommit: 0 @@session.autocommit: 0
test		
begin;	begin;	初始状态查询结果一样 -- select * from tbl;

		+----+------+--------+------+ | id | name | acc_no | amount | +----+------+--------+------+ | 1 | yan | 321 | 100 | +----+------+--------+------+ 1 row in set (0.00 sec)
update tbl set amount=amount+200 where acc_no=321; -- 资金转入200		账户中增加200
	select * from tbl;	在session 2中查询账户金额，金额未发生变化 s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec)
commit; -- 提交事务，确认向该账户中转入200		
	select * from tbl;	待事务1提交之后，再次在事务2中发出查询，发现两次查询的结果**相同**。 s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec) s2> select 'after s1 commit..'\G *************************** 1. row *** after s1 commit..: after s1 commit.. 1 row in set (0.00 sec) s2> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec) --> 此时，并未遇到"Non-Repeatable Read"问题。

Case 5: Phantom Read, REPEATABLE READ

隔离级别设定为"REPEATABLE READ"，测试幻读问题

在事务1查询这一行，然后在事务2中插入新的一行，接着在事务1中重复刚才的读取操作，

session 1	session 2	memo
init		
set global tx_isolation = **'REPEATABLE-READ'**; set global autocommit=0; drop table if exists tbl; create table if not exists tbl(id int primary key, name varchar(20), acc_no int, amount int); insert tbl select 1,'yan',321,100; commit;		set isolation level off auto commit create test table init table data
mysql -uroot sbtest --prompt='s1> '	mysql -uroot sbtest --prompt='s2> '	login
select @@global.tx_isolation, @@session.tx_isolation\G	select @@global.tx_isolation, @@session.tx_isolation\G	check isolation level -- @@global.tx_isolation: **REPEATABLE-READ** @@session.tx_isolation: **REPEATABLE-READ**
select @@global.autocommit, @@session.autocommit\G	select @@global.autocommit, @@session.autocommit\G	check auto commit -- @@global.autocommit: 0 @@session.autocommit: 0
test		
begin;	begin;	初始状态查询结果一样 -- select * from tbl; +----+------+--------+------+ | id | name | acc_no | amount | +----+------+--------+------+ | 1 | yan | 321 | 100 | +----+------+--------+------+ 1 row in set (0.00 sec)
select * from tbl where name = 'yan'; -- 在事务1中读取某行数据		Session1中第一次查询 

		记录
	insert into tbl values (2, 'yan', '123', '2000');	在事务2在新插入一行数据

select * from tbl where name = 'yan';
--
在事务1中再次读取该行数据，数据无变化
		Session1中查询的结果并没有发生变化
		`s1> select * from tbl where name = 'yan';`
		`+----+------+--------+--------+`
		`| id | name | acc_no | amount |`
		`+----+------+--------+--------+`
		`| 1 | yan | 321 | 100 |`
		`+----+------+--------+--------+`
		`1 row in set (0.00 sec)`
		--
		T1, 2nd: MVCC快照，数据是事务开始时的记录

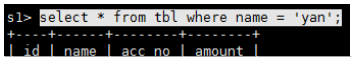
	commit; -- 提交事务2	

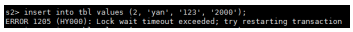
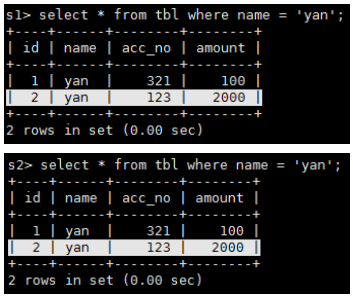
select * from tbl where name = 'yan';
--
在事务1中第3次读取该行数据，数据无变化
		Session1中查询的结果并没有发生变化
		`s1> select * from tbl where name = 'yan';`
		`+----+------+--------+--------+`
		`| id | name | acc_no | amount |`
		`+----+------+--------+--------+`
		`| 1 | yan | 321 | 100 |`
		`+----+------+--------+--------+`
		`1 row in set (0.00 sec)`
		--
		T1, 3rd: 同上
		--> 并没有出现幻读现象。

commit; -- 提交事务1		

select * from tbl where name = 'yan';
--
在事务1中第4次读取该行数据，数据发生变化，可以看到事务2新增的数据
		Session1中查询的结果发生变化
		`s1> select * from tbl where name = 'yan';`
		`+----+------+--------+--------+`
		`| id | name | acc_no | amount |`
		`+----+------+--------+--------+`
		`| 1 | yan | 321 | 100 |`
		`| 2 | yan | 123 | 2000 |`
		`+----+------+--------+--------+`
		`2 rows in set (0.00 sec)`
		--
		T1, 4nd: 此时事务1,2均已提交，在Session1中得到的查询结果其实是两个事务提交成功之后的结果，所以会查询到2条记录。

Case 6: Phantom Read，SERIALIZABLE

session 1	session 2	memo
init		
set global tx_isolation = **'SERIALIZABLE'**; set global autocommit=0; drop table if exists tbl; create table if not exists tbl(id int primary key, name varchar(20), acc_no int, amount int); insert tbl select 1,'yan',321,100; commit;		set isolation level off auto commit create test table init table data
mysql -uroot sbtest --prompt='s1> '	mysql -uroot sbtest --prompt='s2> '	login
select @@global.tx_isolation, @@session.tx_isolation\G	select @@global.tx_isolation, @@session.tx_isolation\G	check isolation level -- @@global.tx_isolation: **REPEATABLE-READ** @@session.tx_isolation: **REPEATABLE-READ**
select @@global.autocommit, @@session.autocommit\G	select @@global.autocommit, @@session.autocommit\G	check auto commit -- @@global.autocommit: 0 @@session.autocommit: 0
test		
begin;	begin;	初始状态查询结果一样 -- select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+ 1 row in set (0.00 sec)
select * from tbl where name = 'yan';		Session1中第一次查询 

		1行，1st. 原始数据，只有1条记录
	insert into tbl values (2, 'yan', '123', '2000'); -- 事务2会一直等待，直到事务1提交，或者锁超时	在事务2在新插入一行数据 -- 
commit; -- 提交事务1		待事务1提交成功后，事务2中的insert才能执行成功
	commit; -- 提交事务2	事务2提交成功，在两个Session中都可以查询到新行 

事务2的插入操作会等待事务1的锁

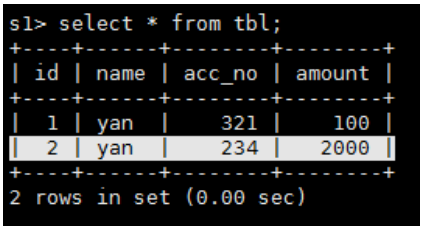
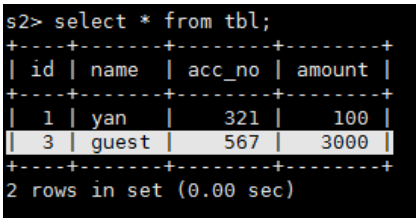
```
select * from information_schema.INNODB_LOCKS\G
*************************** 1. row ***************************
    lock_id: 83039:1440:3:1
lock_trx_id: 83039
  lock_mode: X
  lock_type: RECORD
lock_table: `sbtest`.`tbl`
lock_index: PRIMARY
lock_space: 1440
  lock_page: 3
   lock_rec: 1
  lock_data: supremum pseudo-record
*************************** 2. row ***************************
    lock_id: 421175108206856:1440:3:1
lock_trx_id: 421175108206856
  lock_mode: S
  lock_type: RECORD
lock_table: `sbtest`.`tbl`
lock_index: PRIMARY
lock_space: 1440
  lock_page: 3
   lock_rec: 1
  lock_data: supremum pseudo-record
2 rows in set (0.00 sec)
```

```
                    trx_state: LOCK WAIT
                  trx_started: 2019-06-13 12:13:27
        trx_requested_lock_id: 83039:1440:3:1
             trx_wait_started: 2019-06-13 12:13:27
                   trx_weight: 2
          trx_mysql_thread_id: 672
                    trx_query: insert into tbl values (2, 'yan', '123', '2000')
          trx_operation_state: inserting
            trx_tables_in_use: 1
            trx_tables_locked: 1
             trx_lock_structs: 2
        trx_lock_memory_bytes: 1136
              trx_rows_locked: 1
            trx_rows_modified: 0
      trx_concurrency_tickets: 0
          trx_isolation_level: SERIALIZABLE
            trx_unique_checks: 1
       trx_foreign_key_checks: 1
trx_last_foreign_key_error: NULL
          trx_is_read_only: 0
trx_autocommit_non_locking: 0
*************************** 2. row ***************************
                       trx_id: 421175108206856
                    trx_state: RUNNING
                  trx_started: 2019-06-13 12:09:56
        trx_requested_lock_id: NULL
             trx_wait_started: NULL
                   trx_weight: 2
          trx_mysql_thread_id: 671
                    trx_query: NULL
          trx_operation_state: NULL
            trx_tables_in_use: 0
            trx_tables_locked: 1
             trx_lock_structs: 2
        trx_lock_memory_bytes: 1136
              trx_rows_locked: 2
            trx_rows_modified: 0
      trx_concurrency_tickets: 0
          trx_isolation_level: SERIALIZABLE
            trx_unique_checks: 1
       trx_foreign_key_checks: 1
trx_last_foreign_key_error: NULL
          trx_is_read_only: 0
trx_autocommit_non_locking: 0
2 rows in set (0.00 sec)
```

事务1开始时读取到1条数据，待事务2插入一条新数据并提交成功后，事务1依然可以查询到1条数据，
但是当事务1做更新操作后，却发现将事务2中新插入的数据也一并更新，
如果事务1回滚，那么数据恢复到事务2提交后，
如果事务1提交，那么提交成功，并且将事务2中的新数据一并更新。

session 1	session 2	memo
init		
set global tx_isolation = '**REPEATABLE-READ**'; set global autocommit=0; drop table if exists tbl; create table if not exists tbl(id int primary key, name varchar(20), acc_no int, amount int); insert tbl select 1,'yan',321,100; commit;		set isolation level off auto commit create test table init table data
mysql -uroot sbtest --prompt='s1> '	mysql -uroot sbtest --prompt='s2> '	login
select @@global.tx_isolation, @@session.tx_isolation\G	select @@global.tx_isolation, @@session.tx_isolation\G	check isolation level -- @@global.tx_isolation: **REPEATABLE-READ** @@session.tx_isolation: **REPEATABLE-READ**
select @@global.autocommit, @@session.autocommit\G	select @@global.autocommit, @@session.autocommit\G	check auto commit -- @@global.autocommit: 0 @@session.autocommit: 0
test		
begin;	begin;	初始状态查询结果一样 -- select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | +----+------+--------+--------+

-- trx_id: 83349		show engine innodb status\G ---TRANSACTION **83349**, ACTIVE 10 sec 1 lock struct(s), heap size 1136, 0 row lock(s), undo log entries 1 MySQL thread id 42, OS thread handle 140214714943232, query id 274 localhost root Trx read view will not see trx with id >= 83349, sees < 83349
	insert into tbl values (3,'guest',567,3000); -- trx_id: 83354	---TRANSACTION **83354**, ACTIVE 84 sec 1 lock struct(s), heap size 1136, 0 row lock(s), undo log entries 1 MySQL thread id 44, OS thread handle 140214714943232, query id 279 localhost root Trx read view will not see trx with id >= 83349, sees < 83349 ---TRANSACTION 83349, ACTIVE 86 sec 1 lock struct(s), heap size 1136, 0 row lock(s), undo log entries 1 MySQL thread id 42, OS thread handle 140214714943232, query id 278 localhost root Trx read view will not see trx with id >= 83349, sees < 83349
s1> select * from tbl; +----+------+--------+--------+ | id | name | acc_no | amount | +----+------+--------+--------+ | 1 | yan | 321 | 100 | | 2 | yan | 234 | 2000 | +----+------+--------+--------+ 2 rows in set (0.01 sec)	s2> select * from tbl; +----+-------+--------+--------+ | id | name | acc_no | amount | +----+-------+--------+--------+ | 1 | yan | 321 | 100 | | 3 | guest | 567 | 3000 | +----+-------+--------+--------+ 2 rows in set (0.00 sec)	分别在两个事务中查询全表，得到不同的结果

	commit; Query OK, 0 rows affected (0.00 sec) -- 事务2提交成功	提交事务2 -- ---TRANSACTION **83349**, ACTIVE 191 sec 1 lock struct(s), heap size 1136, **0 row lock(s)**, undo log entries 1 MySQL thread id 42, OS thread handle 140214714943232, query id 278 localhost root Trx read view will not see trx with id >= 83349, sees < 83349
		分别在两个事务中查询全表，得到结果与事务2提交前一致
s1> **update tbl set amount=amount+1;** Query OK, 3 rows affected (0.00 sec) Rows matched: 3 Changed: 3 Warnings: 0		此时，在事务1中对数据进行变更 -- ---TRANSACTION 83349, ACTIVE 473 sec 2 lock struct(s), heap size 1136, **3 row lock(s),** undo