H 03: Isolation Levels

Submitted by: Bryan Alvarez, Sarah Jumilla, Sehyun Park, Shawn Aberin

I. Abstract

The group conducted an experiment based on the four database isolation levels; Read Uncommitted, Read Committed, Repeatable Read, and Serializable. To conduct this experiment the group decided to do one database isolation level each and get results and comparisons from concurrency problems; Dirty Read, Non-repeatable Read, and Phantom Reads. The group used the same transaction tables as the ones in III.

II. Introduction

The objective of this paper is to experiment with the effects of 4 different database isolation levels; read uncommitted, read committed, repeatable read, and serializable, in maintaining the atomicity, consistency, isolation, and durability (ACID) property of the transactions. A multiple experiment for each of the isolation levels will be conducted which will replicate various concurrency problems and investigate how the different isolation levels can be utilized to resolve concurrency issues.

III. Schema used for the experiment

The demonstration schema used two columns. An auto incrementing column of type INT was used as the index and primary key. The other column was the amount of type INT. The table was created with the following query:

CREATE TABLE Accounts(id INT AUTO_INCREMENT PRIMARY KEY, amount INT)

IV. Read Uncommitted

"READ UNCOMMITTED" is the lowest isolation level defined by the SQL standard. In this isolation level, transactions are not isolated from each other, meaning that one transaction can read uncommitted changes made by other concurrent transactions. This level of isolation provides the highest level of concurrency but comes with the risk of dirty reads, non-repeatable reads, and phantom reads.

The following experiments for Read Uncommitted were performed using MYSQL Workbench.

At the beginning of each transaction for this experiment, the following query was applied to initialize the isolation level;

SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

The keyword *Session* sets the isolation level of the database to Read Uncommitted for the current experiment session, reducing the redundancy of having to rewrite the query to reset the isolation level for each transaction that will be performed for this experiment

A. Dirty Read

	T1	T2	Comments
t1	START TRANSACTION		Start of Transaction 1
t2	INSERT INTO accounts (id, amount) VALUES (1, 500);		Write()
t3	UPDATE accounts SET amount = amount + 100 WHERE id = 1		Write()
t4		START TRANSACTION	Start of Transaction 2
t5		SELECT * FROM accounts WHERE id = 1;	Read() - Dirty Read
t6		COMMIT	Commit Transaction 2
t7	COMMIT		Commit Transaction 1

Table 1. Read Uncommitted Dirty Read Query

In Table 1, there are two transactions and transaction 1 initiates first. Within Transaction 1, an UPDATE operation is executed on the accounts table. The amount for the row with id equal to 1 is modified to 500. Then a 100 will be added to id = 1. The changes made by Transaction 1 are committed, making the modification (amount = 100) permanent in the accounts table. Simultaneously, Transaction 2 is initiated using START TRANSACTION;. It starts concurrently with Transaction 1. Within Transaction 2, a SELECT operation is performed to read data from the accounts table where id is 1. Due to the READ UNCOMMITTED isolation level, Transaction 2 reads uncommitted data. It sees the modified value (amount = 100) made by Transaction 1 before Transaction 1 has committed. Transaction 2 commits, finalizing its changes. Transaction 1 continues and commits. The modification made in Transaction 1 (amount = 100) is now permanent in the accounts table. In "READ UNCOMMITTED," a transaction can read uncommitted changes made by other transactions, allowing it to see data that might be rolled back later.

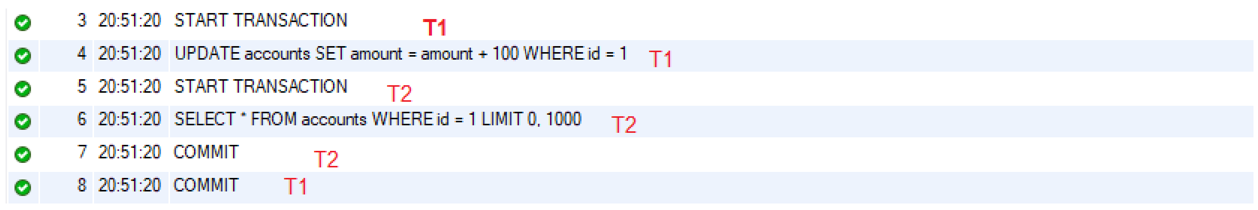
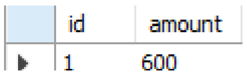

```
3  20:51:20  START TRANSACTION                                       T1
4  20:51:20  UPDATE accounts SET amount = amount + 100 WHERE id = 1  T1
5  20:51:20  START TRANSACTION                                       T2
6  20:51:20  SELECT * FROM accounts WHERE id = 1 LIMIT 0, 1000       T2
7  20:51:20  COMMIT                                                  T2
8  20:51:20  COMMIT                                                  T1
```

Figure 1. Read Uncommitted Snippet

In this scenario, Transaction 2 reads the uncommitted data made by Transaction 1, demonstrating a dirty read. The COMMIT in Transaction 2 allows it to complete before Transaction 1 commits, illustrating the concurrent nature of the transactions.



id	amount
1	600

Figure 2. Read Uncommitted Query Result

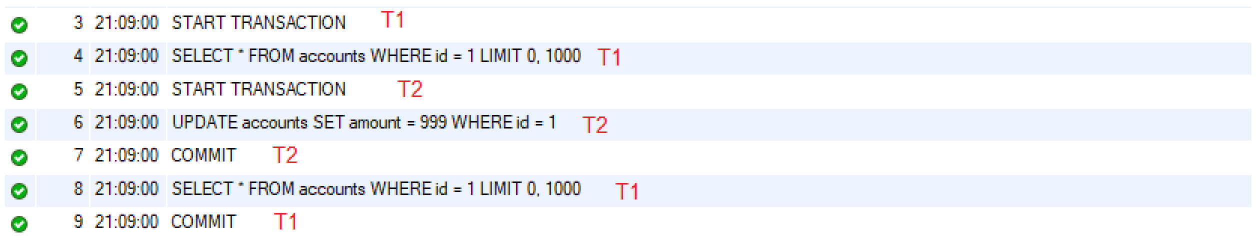
An amount of 600 in id = 1 should be the result of this query.

B. Non-repeatable Read

	T1	T2	Comments
t1	START TRANSACTION		Start of Transaction 1
t2	SELECT * FROM accounts WHERE id = 1; (Initial read)		Read() - Non-repeatable Read
t3		START TRANSACTION	Start of Transaction 2
t4		UPDATE accounts SET amount = 999 WHERE id = 1;	Write()
t5		COMMIT	Commit Transaction 2
t6	SELECT * FROM accounts WHERE id = 1;		Read() - Non-repeatable Read
t7	COMMIT		Commit Transaction 1

Table 2 . Read Uncommitted Non-repeatable Read Query

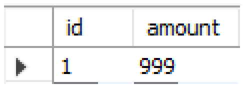
In Table 2, there are two transactions and transaction 1 initiates first. Transaction 1 performs an initial read from the accounts table where id is 1. Transaction 2 begins, executed concurrently with Transaction 1. Transaction 2 updates the amount in the accounts table where id is 1 to 999. Transaction 2 commits. This change is now visible to other transactions. Transaction 1, after the initial read, continues and performs another read from the accounts table where id is 1. Transaction 1 commits. Any changes made by Transaction 1 are now permanent. In "READ UNCOMMITTED," a transaction is not protected from changes made by other transactions, leading to non-repeatable reads.



```
✓  3  21:09:00  START TRANSACTION   T1
✓  4  21:09:00  SELECT * FROM accounts WHERE id = 1 LIMIT 0, 1000   T1
✓  5  21:09:00  START TRANSACTION   T2
✓  6  21:09:00  UPDATE accounts SET amount = 999 WHERE id = 1   T2
✓  7  21:09:00  COMMIT   T2
✓  8  21:09:00  SELECT * FROM accounts WHERE id = 1 LIMIT 0, 1000   T1
✓  9  21:09:00  COMMIT   T1
```

Figure 3. Non-repeatable Read Snippet

In a non-repeatable read scenario, Transaction 1 reads a row, then Transaction 2 updates that row, and finally, Transaction 1 reads the same row again but observes the change made by Transaction 2. This demonstrates the non-repeatable nature of the read, as the data can change within the same transaction due to the actions of another concurrent transaction.



	id	amount
▶	1	999

Figure 4. Non-repeatable Query Result

An amount of 999 in id = 1 should be the result of this query.

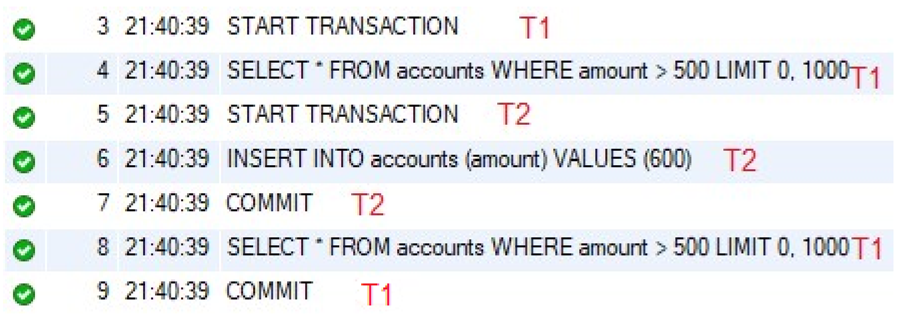
C. Phantom Reads

	T1	T2	Comments
t1	START TRANSACTION		Start of Transaction 1
t2	SELECT * FROM accounts WHERE amount > 500;		Read() - Phantom Read
t3		START TRANSACTION	Start of Transaction 2
t4		INSERT INTO accounts (amount) VALUES (600);	Write()
t5		COMMIT	Commit Transaction 2
t6	SELECT * FROM accounts WHERE amount > 500;		Read() - Phantom Read
t7	COMMIT		Commit Transaction 1

Table 3. Read Uncommitted Phantom Read Query

In Table 3, there are two transactions and transaction 1 initiates first. Transaction 1 begins and reads rows from the accounts table where ‘amount’ is greater than 500. Transaction 2 then begins, executed concurrently with Transaction 1. Transaction 2 inserts a new row into the accounts table with an amount of 600. Transaction 2 commits.

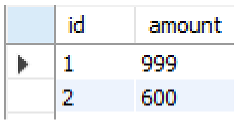
The newly inserted row is now permanent. Transaction 1, after the initial read, continues and performs another read from the accounts table where ‘amount’ is greater than 500. Transaction 1 commits. Any changes made by Transaction 1 are now permanent. In "READ UNCOMMITTED," a transaction can see new rows inserted by other transactions between its reads, resulting in phantom reads.



```
✓ 3 21:40:39 START TRANSACTION                                        T1
✓ 4 21:40:39 SELECT * FROM accounts WHERE amount > 500 LIMIT 0, 1000  T1
✓ 5 21:40:39 START TRANSACTION                                        T2
✓ 6 21:40:39 INSERT INTO accounts (amount) VALUES (600)               T2
✓ 7 21:40:39 COMMIT                                                   T2
✓ 8 21:40:39 SELECT * FROM accounts WHERE amount > 500 LIMIT 0, 1000  T1
✓ 9 21:40:39 COMMIT                                                   T1
```

Figure 5. Phantom Read Snippet

The phantom read occurs because Transaction 1 sees additional rows in the result set during the second read due to new rows inserted by another concurrent transaction (Transaction 2). The "phantom" rows appear as if they've materialized between the two reads, influencing the outcome of the second read in Transaction 1.



id	amount
1	999
2	600

Figure 6. Phantom Read Query Result

Due to phantom read a new row with id =2 was inserted with an amount = 600.

It's essential to use "READ UNCOMMITTED" with caution, as it might lead to unexpected results and data integrity issues. This isolation level is often suitable for scenarios where high concurrency is crucial, and the trade-off for potential inconsistencies is acceptable. However, in many applications, higher isolation levels like "READ COMMITTED," "REPEATABLE READ," or "SERIALIZABLE" are preferred to ensure a higher level of data integrity.

V. Read Committed

Read Committed is the third highest isolation level which prevents dirty reads but allows non-repeatable reads and phantom read concurrency problems.

The following experiments replicating the three concurrency problems for Read Committed isolation level was performed in MySQL 8.0 Command Line Client (CLC). The CLC was chosen to be worked with this experiment as all of the cases for this experiment involve observing the behavior of multiple concurrent transactions by a single user. By utilizing CLC, the group will be able to observe the behavior of the transactions line by line without the usage of methods such as SLEEP() to postpone each query, as CLC will be fed with a single transaction line at the time instead of running the entire test script at once.

At the beginning of each transaction for this experiment, the following query was applied to initialize the isolation level;

SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

The keyword *Session* sets the isolation level of the database to Read Committed for the current experiment session, reducing the redundancy of having to rewrite the query to reset the isolation level for each transaction that will be performed for this experiment.

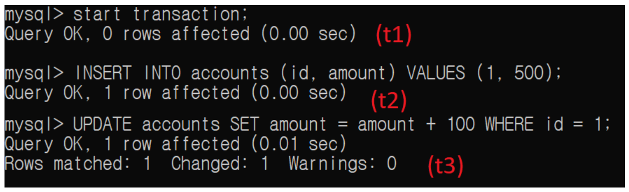
The transaction table used for this isolation level is equivalent to the one's used for the Read Uncommitted portion of the paper, thus the tables will not be rewritten but rather referred to accordingly.

A. Dirty Read

Dirty read refers to a situation where one transaction reads uncommitted change made by another transaction. To replicate this scenario, the following transactions were created (Table 1).

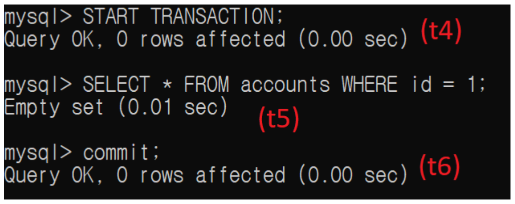
In the Table 1 mentioned above, there are two transactions involved where T1 performs INSERT and UPDATE on the database, while T2 performs a SELECT from the same database. Notice how T1 starts its transaction in t1 but doesn't commit until t7, while T2 starts its transaction at t4 and commits on t6 before T1 does. This is an example of dirty read where T2 is trying to read uncommitted changes made by T1, and according to the behavior of Read Committed isolation level, as the name suggests it should not be able to read the uncommitted changes made by the previous transactions.

The query was replicated in CLC and its behavior was observed. The following scenario was simulated with the following transactions (Table 1). This experiment aims to investigate the behavior of the Read Commit isolation level with respect to dirty reads and observe its safety in maintaining data consistency.



```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)  (t1)

mysql> INSERT INTO accounts (id, amount) VALUES (1, 500);
Query OK, 1 row affected (0.00 sec)  (t2)

mysql> UPDATE accounts SET amount = amount + 100 WHERE id = 1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0  (t3)
```

Figure 7. T1 Snippet

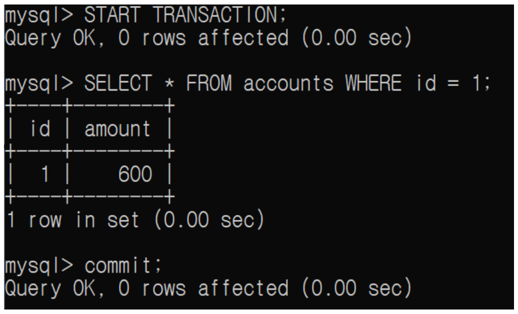


```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec) (t4)

mysql> SELECT * FROM accounts WHERE id = 1;
Empty set (0.01 sec)  (t5)

mysql> commit;
Query OK, 0 rows affected (0.00 sec) (t6)
```

Figure 8. T2 Snippet

Notice how in Figure 7, the T1 was performed with both INSERT and UPDATE to the accounts table, and Figure 8 occurred concurrently with the T1 before it was able to commit the changes. As shown in Figure 8, the T2 was not able to read the changes made by the T1, thus showing *Empty Set* as a result since the accounts table was initially empty at the beginning of the transaction.

For T2 to read the changes made by T1 in Read Committed isolation level, the user simply has to commit the T1 then perform the read on T2. Figure 14 shows that after committing the transaction in T1, the T2 was then able to read the changes, showing the behavior of the Read Commit isolation level for the dirty read concurrency problem. The only isolation level where it could read the uncommitted transaction would be the Read Uncommitted isolation level. If the experiment was to be initialized to Read Uncommitted, the result of Figure 8 would have shown the inserted and updated accounts table instead of the *Empty Set*.



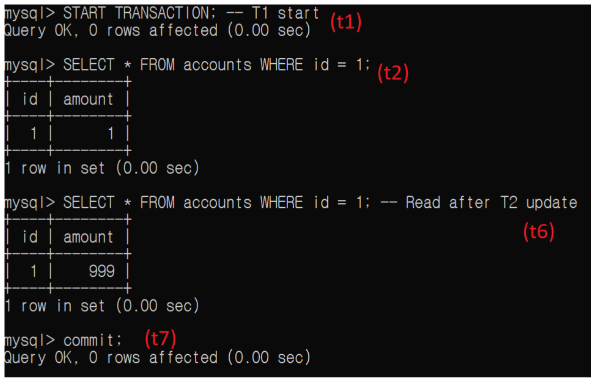
```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    600 |
+----+--------+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

Figure 14. New T2 Snippet after T1 Commit

B. Non-repeatable Read

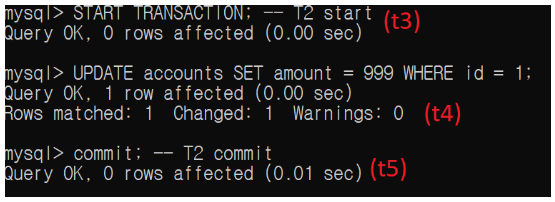
Non-repeatable Read refers to a situation where a transaction reads a data, another transaction modifies it, and the first transaction is able to observe the changes made by the second transaction. The following scenario was simulated with the following transactions (Table 2).

In the given transaction, T1 reads the data twice, once in t2 and another one in t6. The first read in t2 reads the unmodified data which were initially present in the database, while the second read in t6 reads the data after it has been modified by another concurrent transaction T2 (Figure 15, 9). This experiment aims to investigate the behavior of the Read Commit isolation level with respect to non-repeatable reads and observe its safety in maintaining data consistency.



```
mysql> START TRANSACTION; -- T1 start  (t1)
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1; (t2)
+----+--------+
| id | amount |
+----+--------+
|  1 |      1 |
+----+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1; -- Read after T2 update
+----+--------+                                                (t6)
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> commit;  (t7)
Query OK, 0 rows affected (0.00 sec)
```

Figure 16. T1 Snippet



```
mysql> START TRANSACTION; -- T2 start (t3)
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE accounts SET amount = 999 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0  (t4)

mysql> commit; -- T2 commit
Query OK, 0 rows affected (0.01 sec) (t5)
```

Figure 9. T2 Snippet

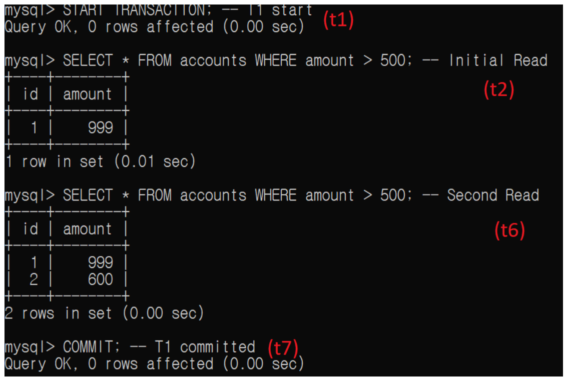
The following figures show that T2 occurred concurrently with the T1 while it hasn't committed yet, allowing T2 to intervene T1 and make changes to the data in between. Due to the property of Read Committed isolation level, T1 was able to observe the UPDATE made by T2 during its active transaction, showing two different results when given the same query in the single instance of a transaction (Figure 16).

This property of Read Committed can be dangerous in multiple factors, causing inconsistency in data and data integrity issues. Figure 16 demonstrates how an equal query was able to return different conclusions within the same transaction, this may lead to application errors and flaws in logic which may lead to problems for users and product owners. This section of the experiment showed the importance of understanding and controlling the isolation level depending on the specific requirement, and how much damage it can cause towards the database when handled incorrectly.

C. Phantom Reads

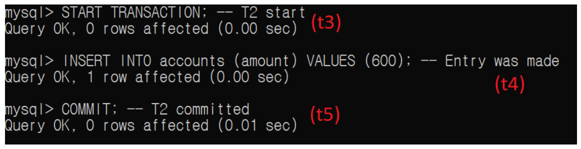
Phantom Reads refers to the situation where an initial transaction performs a range query, another concurrent transaction inserts data within the range, and when the initial transaction repeats the same range query again, it will observe data which was not present during the first range query. The following scenario was simulated with the following transactions (Table 3).

In the given transaction, the T1 performs the range query twice, once in t2 and another one in t6. Range query performed in t2 shows the unmodified initial data from the database, but t6 will be performed after T2 has inserted an entry to the database (Figure 17, 9). This experiment aims to investigate the behavior of the Read Commit isolation level with respect to phantom reads and observe its safety in maintaining data consistency.



```
mysql> START TRANSACTION; -- T1 start
Query OK, 0 rows affected (0.00 sec)  (t1)

mysql> SELECT * FROM accounts WHERE amount > 500; -- Initial Read
+----+--------+
| id | amount |                                      (t2)
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.01 sec)

mysql> SELECT * FROM accounts WHERE amount > 500; -- Second Read
+----+--------+
| id | amount |                                      (t6)
+----+--------+
|  1 |    999 |
|  2 |    600 |
+----+--------+
2 rows in set (0.00 sec)

mysql> COMMIT; -- T1 committed (t7)
Query OK, 0 rows affected (0.00 sec)
```

Figure 17. T1 Snippet



```
mysql> START TRANSACTION; -- T2 start
Query OK, 0 rows affected (0.00 sec)  (t3)

mysql> INSERT INTO accounts (amount) VALUES (600); -- Entry was made
Query OK, 1 row affected (0.00 sec)
                                                        (t4)
mysql> COMMIT; -- T2 committed
Query OK, 0 rows affected (0.01 sec)  (t5)
```

Figure 10. T2 Snippet

The following figures show that T2 occurred concurrently with the T1 while it hasn't committed yet, allowing T2 to insert new data while T1 is ongoing. Due to the property of Read Committed isolation level, T1 was able to observe the INSERT made by T2 during its active transaction, showing two different results when performing the same range query in the single instance of a transaction (Figure 17).

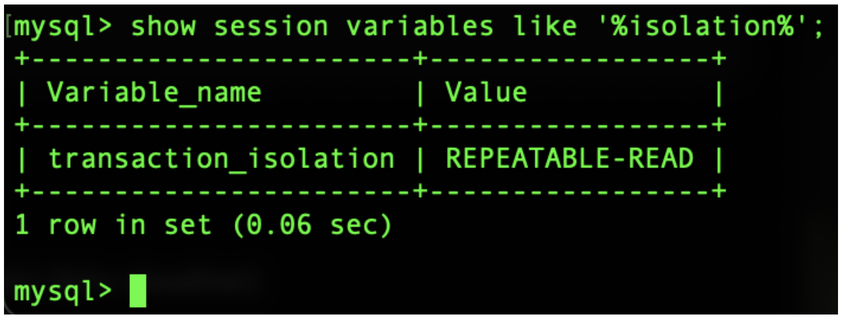
This experiment showed that in Read Commit isolation level, the transaction can observe any new entry to the data when two or more concurrent transactions are performed. As well-demonstrated in Figure 17, it is noticeable that the same range query within the single instance of a transaction showed two different results.

Phantom reads can be especially problematic when performed on calculation-heavy fields such as financial or inventory transactions. The sudden change in active transactions can lead to incorrect calculations and discrepancies in client accounts. To avoid phantom read problems, it is important to understand each of the isolation levels and their behavior to resolve an issue. Utilizing isolation such as serialization where only a single transaction can be made at the time may be a considerable option.

VI. Repeatable Read

The Repeatable Read stands as the second highest, just before serializable. Transactions in this level are prohibited from performing Dirty Read and Non-repeatable Reads but may allow Phantom Reads.

The following experiments were conducted through the MySQL Command Line Client (CLC). According to the MySQL 8.0 Reference Manual's section on Transaction Isolation Levels, the default isolation level in InnoDB is Repeatable Read; hence, during the start of the experiment for this level, no adjustments were made because this default setting was intended for use.



```
mysql> show session variables like '%isolation%';
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.06 sec)

mysql> 
```

Figure 18. Initial Session Transaction Isolation Level

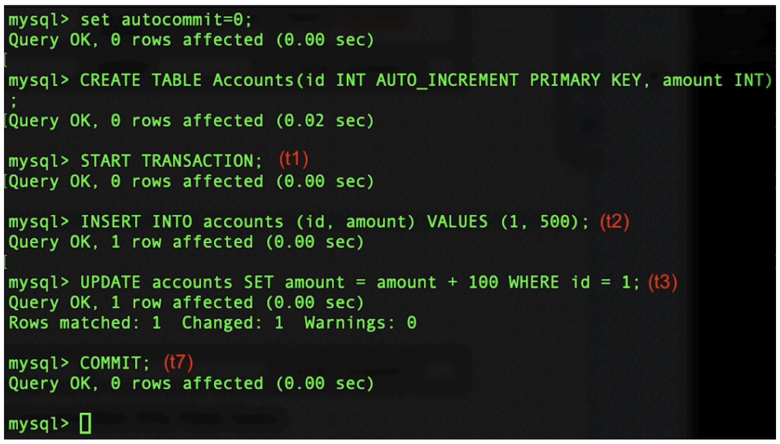
If another isolation level is set, to change to Repeatable Read, at the beginning of each transaction, the following query should be entered to the command line:

SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;

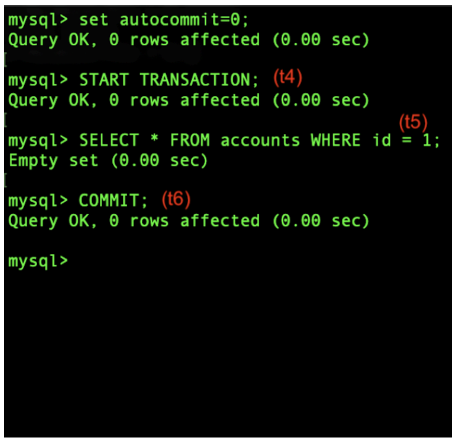
As initially stated, just like for the other isolation levels, for this specific isolation level, the transaction table that will be utilized is the same as the one in the Read Uncommitted section.

A. Dirty Read

To investigate whether repeatable read transactions allow or prevent dirty reads, the sequence outlined in Table 1 was followed. As shown in the figures 11 and 19, two transactions, T1 and T2, were executed. In T1, a row was inserted and updated in the accounts table without being committed yet. Then, in T2, that same table was read. Upon reading, an “Empty set” was returned, specifying that the uncommitted changes in T1 were not read. This indicates that repeatable reads prevent dirty reads from occurring because it does not allow reading uncommitted data in transactions. This reveals the effectiveness of this isolation level in upholding data consistency.



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

mysql> CREATE TABLE Accounts(id INT AUTO_INCREMENT PRIMARY KEY, amount INT)
;
Query OK, 0 rows affected (0.02 sec)

mysql> START TRANSACTION;  (t1)
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO accounts (id, amount) VALUES (1, 500); (t2)
Query OK, 1 row affected (0.00 sec)

mysql> UPDATE accounts SET amount = amount + 100 WHERE id = 1; (t3)
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> COMMIT;  (t7)
Query OK, 0 rows affected (0.00 sec)

mysql> 
```



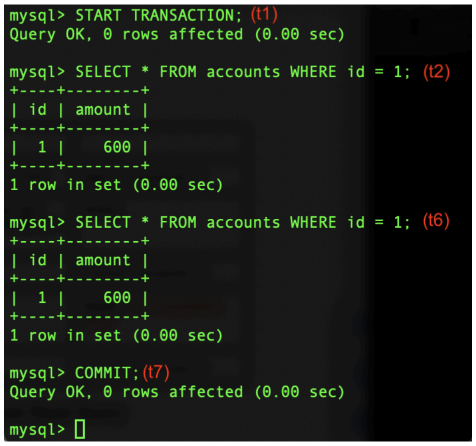
```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION;  (t4)
Query OK, 0 rows affected (0.00 sec)
                                              (t5)
mysql> SELECT * FROM accounts WHERE id = 1;
Empty set (0.00 sec)

mysql> COMMIT;  (t6)
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Figure 19. T1 Snippet

Figure 11. T2 Snippet

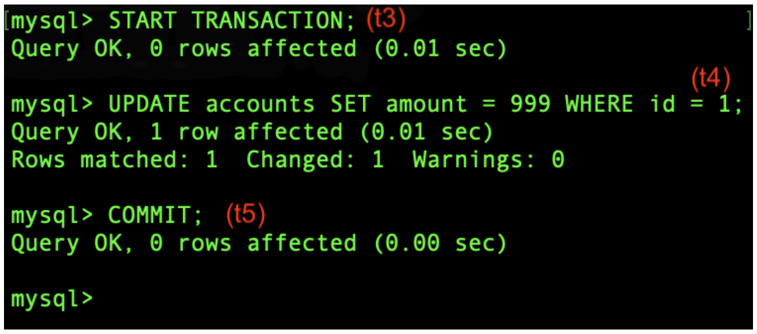
B. Non-repeatable Read

Table 2 was utilized to examine how the repeatable read isolation level handles non-repeatable reads. First, in T1, a read operation on the accounts table was executed. Then, in T2, the accounts table was updated and this change was committed. Upon revisiting T1, another read operation on the same table was performed and despite the committed changes in T2, the data read or retrieved remain unchanged from the initial read operation. Hence, it can be confirmed that non-repeatable reads are not allowed when the isolation level is Repeatable Read. This ensures that the first data read remains consistent throughout its execution.



```
mysql> START TRANSACTION; (t1)
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1; (t2)
+----+--------+
| id | amount |
+----+--------+
|  1 |    600 |
+----+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1; (t6)
+----+--------+
| id | amount |
+----+--------+
|  1 |    600 |
+----+--------+
1 row in set (0.00 sec)

mysql> COMMIT; (t7)
Query OK, 0 rows affected (0.00 sec)

mysql> 
```

Figure 19. T1 Snippet



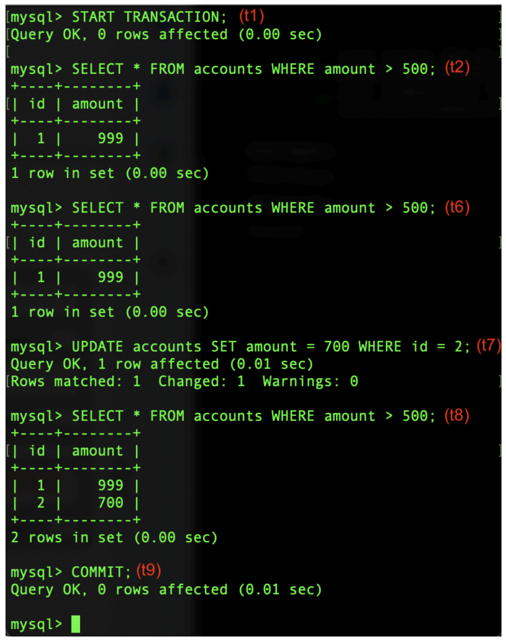
```
mysql> START TRANSACTION; (t3)
Query OK, 0 rows affected (0.01 sec)
                                                   (t4)
mysql> UPDATE accounts SET amount = 999 WHERE id = 1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> COMMIT;  (t5)
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Figure 12. T2 Snippet

C. Phantom Reads

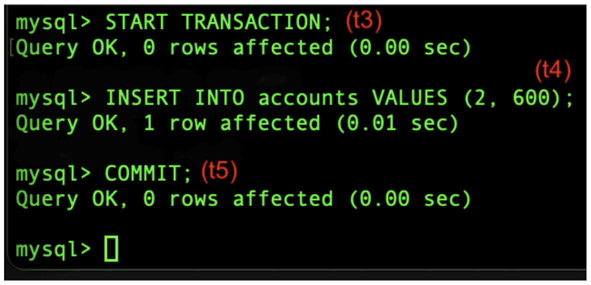
To simulate and evaluate how phantom read is in the repeatable read isolation level, table 3 was first used. In T1, a read operation was performed on the accounts table and then in T2, new data was inserted to the same table and a commit was made. Then, going back to T1, a read operation was performed with a condition that the data recently inserted in T2 satisfies. However, in T1, this new data was not read or did not appear.

But as learned, phantom reads are permissible in repeatable reads. To demonstrate this, an additional step was done which was referenced on the video posted by Sri vlogs on Youtube. This included performing an update operation to the recently inserted row by specifying its id, in T1. Then, a read operation on the table was performed again which now showcases a phantom read as the new row was read and returned. Table 4 shows these additional steps merged with those in table 3.



```
mysql> START TRANSACTION; (t1)
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE amount > 500; (t2)
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE amount > 500; (t6)
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> UPDATE accounts SET amount = 700 WHERE id = 2; (t7)
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM accounts WHERE amount > 500; (t8)
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
|  2 |    700 |
+----+--------+
2 rows in set (0.00 sec)

mysql> COMMIT; (t9)
Query OK, 0 rows affected (0.01 sec)

mysql>
```

Figure 20. T1 Snippet



```
mysql> START TRANSACTION; (t3)
Query OK, 0 rows affected (0.00 sec)
                                                (t4)
mysql> INSERT INTO accounts VALUES (2, 600);
Query OK, 1 row affected (0.01 sec)

mysql> COMMIT; (t5)
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Figure 13. T2 Snippet

	T1	T2	Comments
t1	START TRANSACTION		Start of Transaction 1
t2	SELECT * FROM accounts WHERE amount > 500;		Read()
t3		START TRANSACTION	Start of Transaction 2
t4		INSERT INTO accounts VALUES (2, 600);	Write()
t5		COMMIT	Commit Transaction 2
t6	SELECT * FROM accounts WHERE amount > 500;		Read()
t7	UPDATE accounts SET amount = 700 WHERE id = 2;		Write() - T1 is able to update the row that T2 has just inserted
t8	SELECT * FROM accounts WHERE amount > 500;		Read() - Phantom Read
t9	COMMIT		Commit Transaction 1

Table 4. Repeatable Read Phantom Read Query

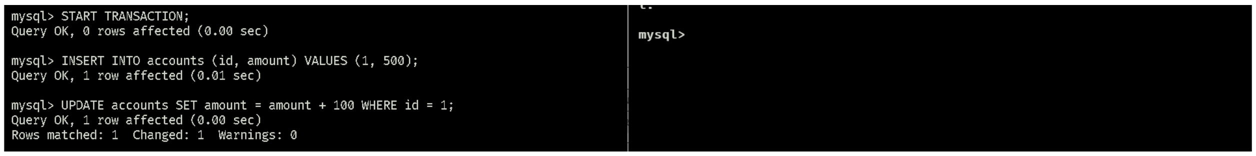
VII. Serializable

Serializable is the highest isolation level. It avoids Dirty Read, Non-repeatable Read, and Phantom Reads since transaction execution looks serial. Concurrent transactions appear not interleaved and seem isolated from another.

Two MySQL CLI clients are used which send transactions concurrently with each other. When replicating this in InnoDB, set autocommit to 0 else each SELECT statement becomes its own transaction that commits itself. *SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE* statement sets the isolation level for the sessions where SERIALIZABLE is the LEVEL.

A. Dirty Read

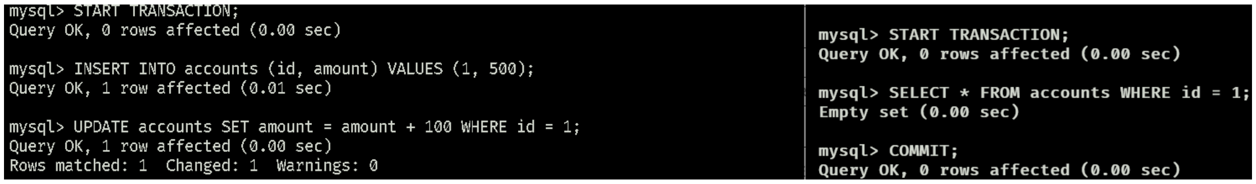
Left user starts the first transaction(T1) that inserts and updates a value into accounts. The right client starts T2 before. T2 does not read uncommitted data hence read is "clean".



```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO accounts (id, amount) VALUES (1, 500);
Query OK, 1 row affected (0.01 sec)

mysql> UPDATE accounts SET amount = amount + 100 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql>
```

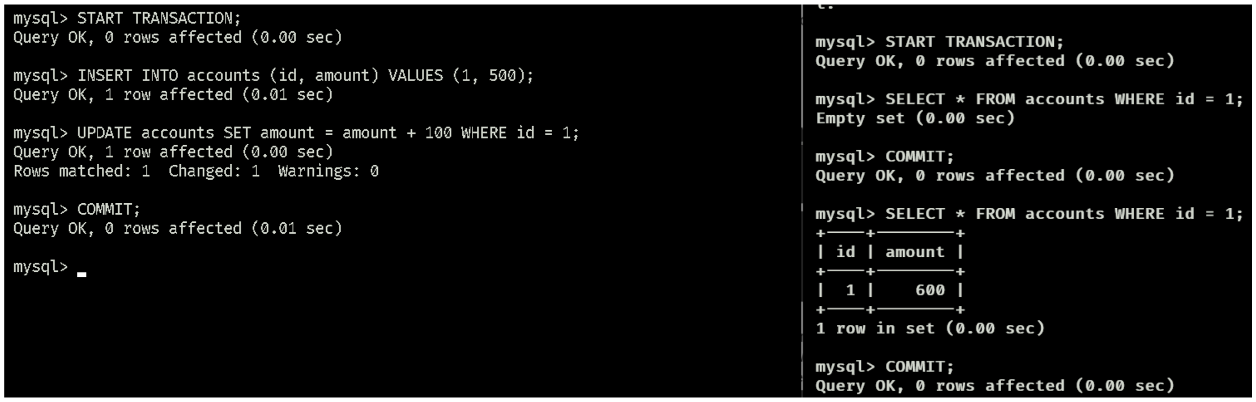
Fig. 21: T1 Start



```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO accounts (id, amount) VALUES (1, 500);
Query OK, 1 row affected (0.01 sec)

mysql> UPDATE accounts SET amount = amount + 100 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
Empty set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Fig. 22: T2 Start and commit. Uncommitted data is unread.



```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO accounts (id, amount) VALUES (1, 500);
Query OK, 1 row affected (0.01 sec)

mysql> UPDATE accounts SET amount = amount + 100 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
Empty set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    600 |
+----+--------+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

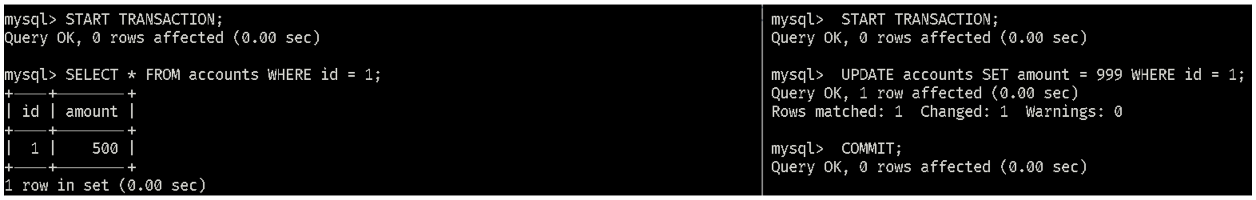
Fig. 23: Commit T1. Right client Read() returns updated value.

Any read done after T2 commits will see the update.

B. Non-repeatable Read

Two clients transact concurrently. In Fig 24, Left client executes T1; another executes T2. Queries are executed as seen in the table 2. T1 reads from the database first then

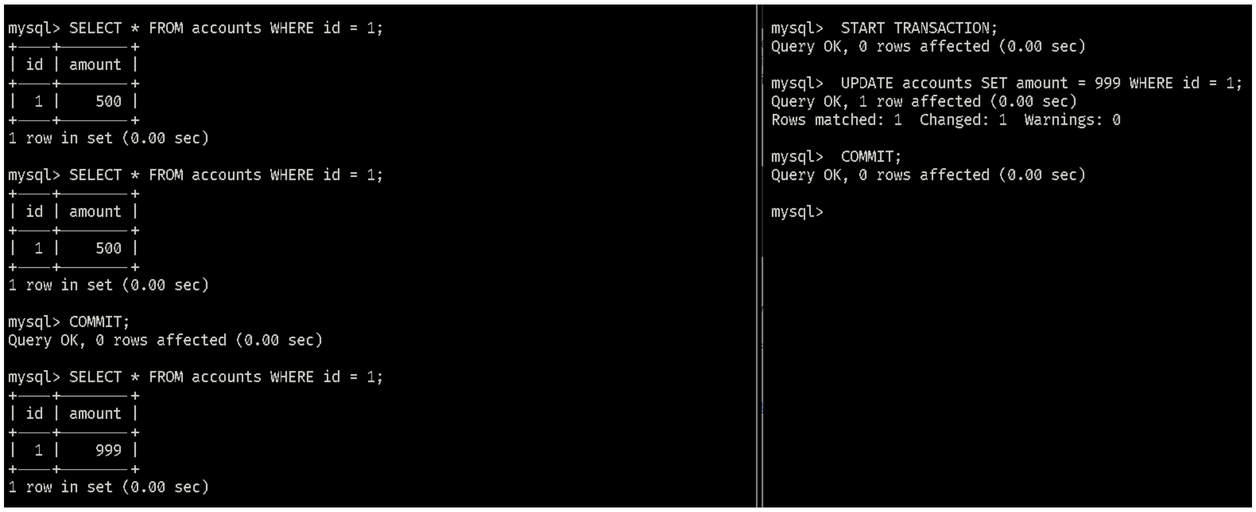
T2 concurrently updates it and commits its changes. But the update is unseen when resuming T1 hence the read is repeatable.



```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    500 |
+----+--------+
1 row in set (0.00 sec)
```

```
mysql>  START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql>  UPDATE accounts SET amount = 999 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>  COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

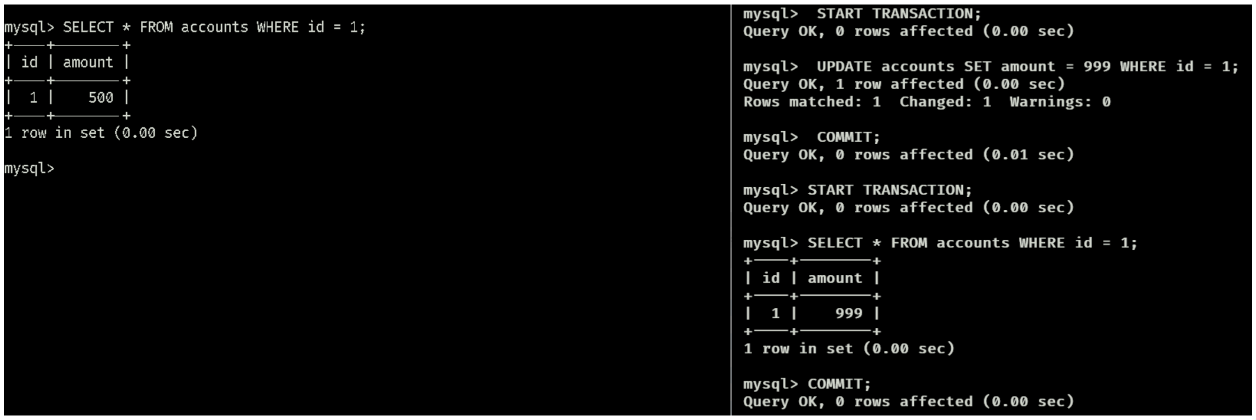
Fig. 24: T1 starts in the right client, T2 starts and commits on the left



```
mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    500 |
+----+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    500 |
+----+--------+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)
```

```
mysql>  START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql>  UPDATE accounts SET amount = 999 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>  COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Fig. 25: With Transactions after T2 in the left client



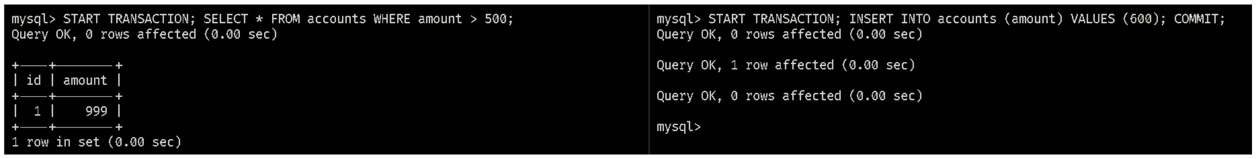
```
mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    500 |
+----+--------+
1 row in set (0.00 sec)

mysql>
```

```
mysql>  START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql>  UPDATE accounts SET amount = 999 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>  COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE id = 1;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Fig. 26: Transactions after T2 in the right client

C. Phantom Reads

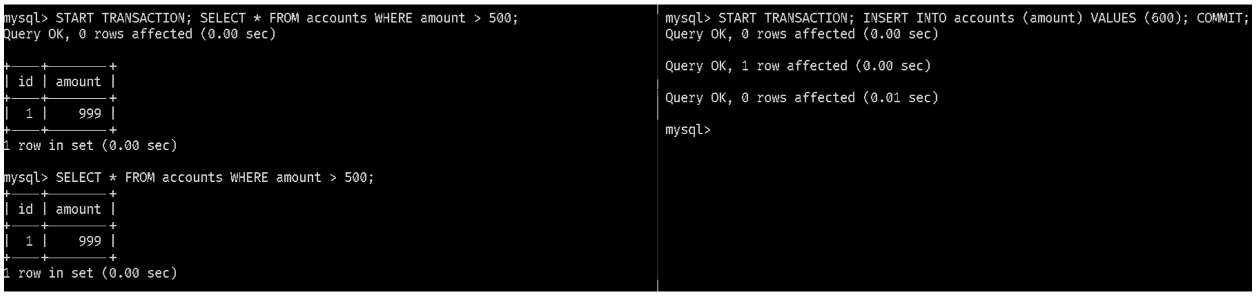
In this level, Phantom Reads are avoided. Testing used two concurrent clients. Query orders are based on table 3. The left client starts reading the database (T1) while the other client starts a transaction (T2) that inserts a new row then commits it in Fig. 27. Continuing T1 after the insert transaction yields the same results as seen in Fig. 28. The new row is not yet read. But if a transaction starts after T2 commits, the new row is seen which is evident in figures 29 and 30.



```
mysql> START TRANSACTION; SELECT * FROM accounts WHERE amount > 500;
Query OK, 0 rows affected (0.00 sec)

+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)
```

```
mysql> START TRANSACTION; INSERT INTO accounts (amount) VALUES (600); COMMIT;
Query OK, 0 rows affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql>
```

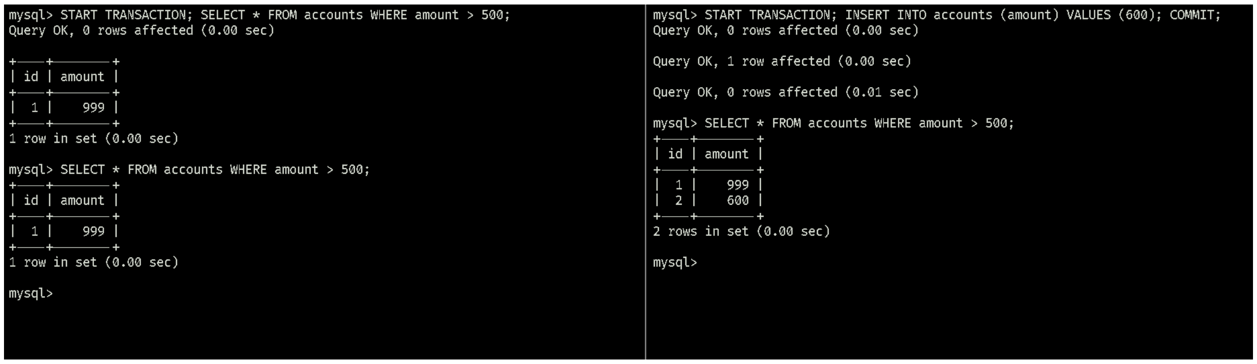
Fig 27: T1 starts in the left, T2 creates a new row in the right



```
mysql> START TRANSACTION; SELECT * FROM accounts WHERE amount > 500;
Query OK, 0 rows affected (0.00 sec)

+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE amount > 500;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)
```

```
mysql> START TRANSACTION; INSERT INTO accounts (amount) VALUES (600); COMMIT;
Query OK, 0 rows affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

mysql>
```

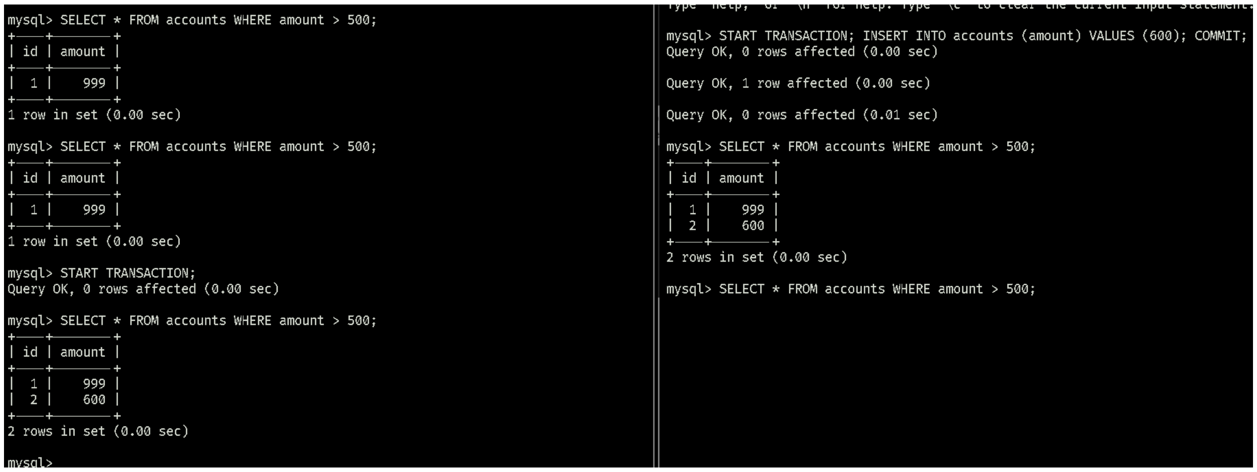
Fig 28: T1 has no Phantom reads



```
mysql> START TRANSACTION; SELECT * FROM accounts WHERE amount > 500;
Query OK, 0 rows affected (0.00 sec)

+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE amount > 500;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql>
```

```
mysql> START TRANSACTION; INSERT INTO accounts (amount) VALUES (600); COMMIT;
Query OK, 0 rows affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM accounts WHERE amount > 500;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
|  2 |    600 |
+----+--------+
2 rows in set (0.00 sec)

mysql>
```

Fig. 29: Client on the right reads after T2



```
mysql> SELECT * FROM accounts WHERE amount > 500;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE amount > 500;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
+----+--------+
1 row in set (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM accounts WHERE amount > 500;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
|  2 |    600 |
+----+--------+
2 rows in set (0.00 sec)

mysql>
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> START TRANSACTION; INSERT INTO accounts (amount) VALUES (600); COMMIT;
Query OK, 0 rows affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM accounts WHERE amount > 500;
+----+--------+
| id | amount |
+----+--------+
|  1 |    999 |
|  2 |    600 |
+----+--------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM accounts WHERE amount > 500;
```

Fig. 30: Client on the left reads after T2

No Phantom reads are evident here. The rows returned are the same. Again this is why the level is called Serializable, the transactions appear as if they were executed one after another independently. The previous transaction is unaffected by the next.

VIII. Why the need?

Each isolation level is a standard way to deal with the Phantom reads, Non-repeatable reads, and Dirty reads, but blocking them has tradeoffs. Isolating Some levels increases performance by sacrificing reliability since more of the three phenomena can appear. Others isolate transactions more for reliability but that costs performance. Developers set isolation levels for their needs and tolerances.

IX. References

https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html
https://stackoverflow.com/questions/73793270/non-repeatable-read-and-phantom-read-occur-with-serializable-isolation-lev
https://www.geeksforgeeks.org/transaction-isolation-levels-dbms/
Deadlocked! (codinghorror.com)
MySQL Isolation Levels: A Guide | Basedash
https://www.youtube.com/watch?v=2QiqVbWK8Mk&ab_channel=Srivlogs

X. Contributions

Name	Documentation	Technical
Bryan Alvarez	25%	25%
Shawn Aberin	25%	25%
Sarah Jumilla	25%	25%
Sehyun Park	25%	25%