1. Introduction to Database Transactions

Database Transactions

A database transaction is a sequence of one or more operations performed as a discrete logical unit of work. These operations may include tasks along with parsing, writing, updating, or deleting records. A transaction is characterized by its potential to ensure the integrity and consistency of certain records, especially in environments where more than one customer accesses and manipulates statistics simultaneously. This is important in scenarios such as banking systems where the accuracy of account balances must be maintained regardless of multiple simultaneous transactions, or in e-commerce platforms where inventory must be accurately current to save overselling.

For example, in a banking facility, a transaction might involve transferring cash from one account to another. If any part of this method fails (e.g., debiting one account to one account but failing on every other), the entire transaction must be rolled back to maintain the integrity of the records. Similarly, in e-commerce, while the buyer places an order, the system should ensure that the inventory is reduced and the order information is recorded correctly. These scenarios underscore the important position of transactions in maintaining reliable and stable database states.

Properties of Transactions (ACID Properties)

Database transactions are governed by the ACID properties: **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These properties collectively ensure that transactions are processed reliably and maintain the integrity of the database.

Atomicity

Atomicity refers to the "all-or-nothing" principle of transactions. This means that a transaction must either be fully completed or not executed at all. If any part of the transaction fails, the entire transaction is rolled back, and the database state remains unchanged. For example, consider a transaction that transfers money from Account A to Account B. The transaction involves two operations: debiting Account A and crediting Account B. If the debiting operation succeeds but the crediting operation fails, the transaction is rolled back, and both accounts remain in their original states, preventing any data corruption.

Consistency

Consistency ensures that a transaction brings the database from one valid state to another, maintaining all predefined rules and integrity constraints. This means that any data written to the database must be valid according to all defined rules, including constraints, cascades, and triggers. For example, in a database managing student records, a consistency rule might ensure that a student's age cannot be a negative number. Transactions must respect this rule, ensuring that any change in the student's record (e.g., updating their age) results in a valid and consistent state.

Isolation

Isolation ensures that transactions occur independently of one another. Even if multiple transactions are executed concurrently, each transaction should be unaware of the others' intermediate states. This is crucial to prevent phenomena such as dirty reads, non-repeatable reads, and phantom reads, which can lead to inaccurate computations. For instance, in an online booking system, two users attempting to book the last available seat at the same time should not interfere with each other's transactions. Isolation mechanisms like locking and concurrency control ensure that transactions are processed in a way that simulates serial execution, thereby maintaining accuracy.

Durability
Durability guarantees that once a transaction has been committed, its changes are permanent, even in the event of a system failure. This means that the results of a committed transaction are stored reliably in non-volatile memory. For example, after a transaction to transfer funds between bank accounts is completed, the new account balances must persist even if the database crashes immediately afterward. Database management systems achieve durability through techniques such as write-ahead logging and checkpointing, which ensure that changes are saved and can be recovered if necessary.

In conclusion, database transactions play a vital role in maintaining data integrity and consistency in multi-user environments. The ACID properties—Atomicity, Consistency, Isolation, and Durability—are fundamental principles that ensure transactions are processed reliably and the database remains in a consistent state, even in the presence of concurrent transactions and potential system failures. Understanding and implementing these properties is crucial for developing robust and reliable database systems.

2. Introduction to Isolation

Defining the Four Phenomena Prohibited by the SQL Standard

Dirty Reads
Dirty reads occur when a transaction reads data that has been modified by another transaction but not yet committed. If the other transaction is rolled back, the data read by the first transaction becomes invalid, leading to inconsistencies.

Non-Repeatable Reads
Non-repeatable reads happen when a transaction reads the same row twice and gets different values each time because another transaction has modified and committed changes to that row in the meantime.

Phantom Reads
Phantom reads occur when a transaction reads a set of rows that satisfy a condition, and another transaction inserts or deletes rows that would satisfy the same condition. When the first transaction re-executes the query, it gets a different set of rows, leading to inconsistencies.

Serialization Anomalies
Serialization anomalies, also known as write skew, happen when the result of executing transactions concurrently is different from the result of executing them sequentially. This anomaly can cause complex inconsistencies that simple locks or checks cannot resolve.

Overview of Isolation Levels

Read Uncommitted
At the Read Uncommitted isolation level, transactions can read data modified by other transactions before they commit, leading to dirty reads. This level provides minimal isolation and is typically used for scenarios where high performance is required, and occasional inconsistencies are acceptable.

Scenario: In a high-frequency trading system, transactions need to access the latest data as quickly as possible, even if it is not yet committed, to make rapid trading decisions.

Read Committed
Read Committed ensures that transactions only see data that has been committed by other transactions, thus preventing dirty reads. However, non-repeatable reads can still occur because other transactions can modify data after it is read but before the current transaction commits.

Scenario: In an online banking system, a user checking their account balance sees only committed transactions, preventing them from seeing inconsistent intermediate states of other transactions.

Repeatable Read
Repeatable Read ensures that if a transaction reads a record twice, it will get the same values both times, preventing non-repeatable reads. However, phantom reads can still occur because other transactions can insert or delete rows that would affect the query results.

Scenario: In an inventory management system, a user generating a report on stock levels will see consistent values for specific items even if other transactions are updating those items concurrently.

Serializable
Serializable is the highest isolation level, ensuring full isolation from other transactions. It prevents all the anomalies: dirty reads, non-repeatable reads, and phantom reads. Transactions are executed in a way that produces the same result as if they were executed serially, one after the other.

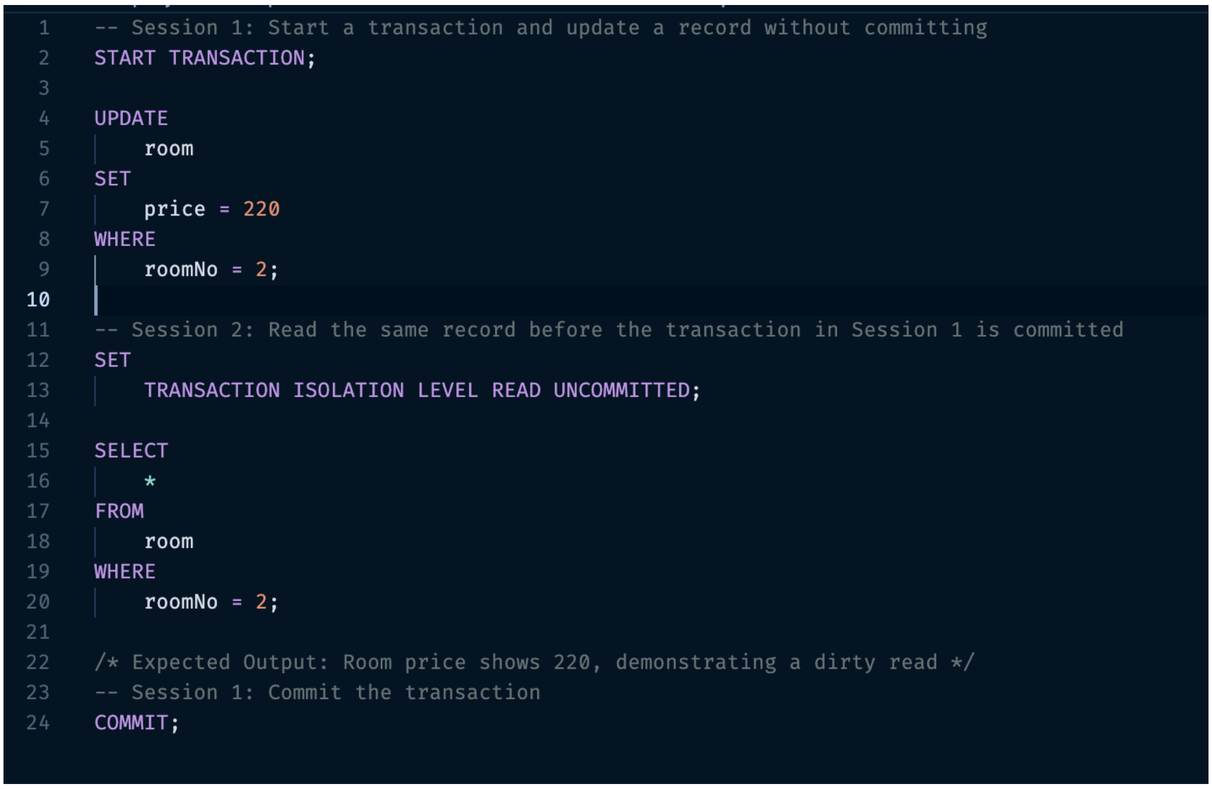
Scenario: In a financial accounting system, ensuring the highest level of data consistency and integrity is crucial, requiring transactions to be fully isolated from each other.

Comparison of Isolation Levels

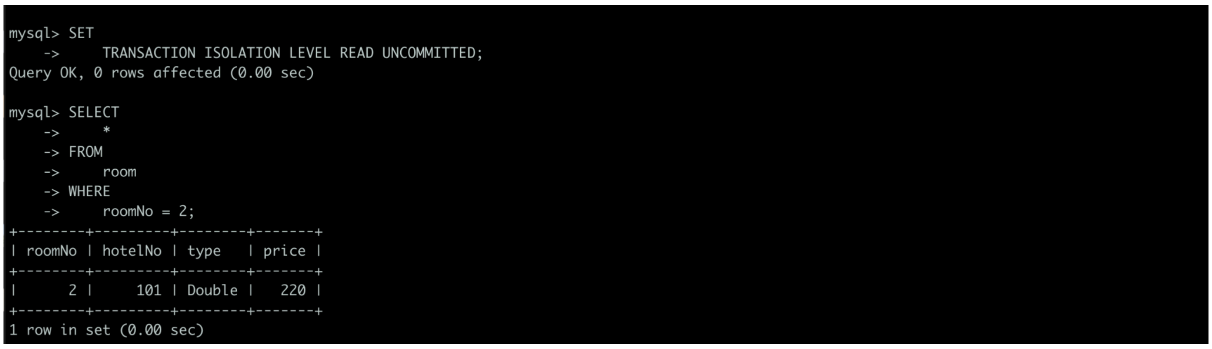
Isolation Level	**Dirty Reads**	**Non-Repeatable Reads**	**Phantom Reads**	**Serialisation Anomalies**
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	Yes	Yes
Serializable	No	No	No	No

3. Practical Demonstration

Read-uncommitted:

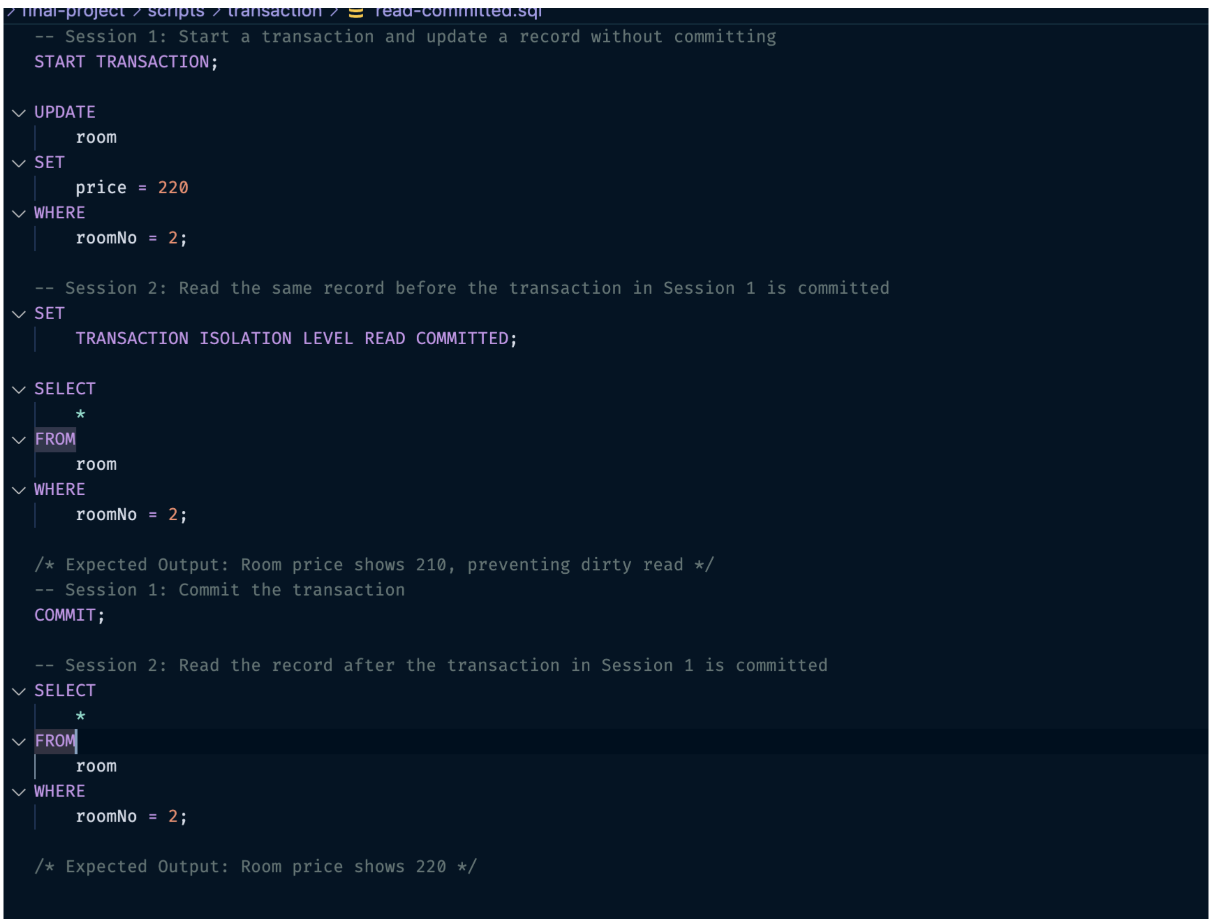


```sql
-- Session 1: Start a transaction and update a record without committing
START TRANSACTION;

UPDATE
    room
SET
    price = 220
WHERE
    roomNo = 2;

-- Session 2: Read the same record before the transaction in Session 1 is committed
SET
    TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT
    *
FROM
    room
WHERE
    roomNo = 2;

/* Expected Output: Room price shows 220, demonstrating a dirty read */
-- Session 1: Commit the transaction
COMMIT;
```

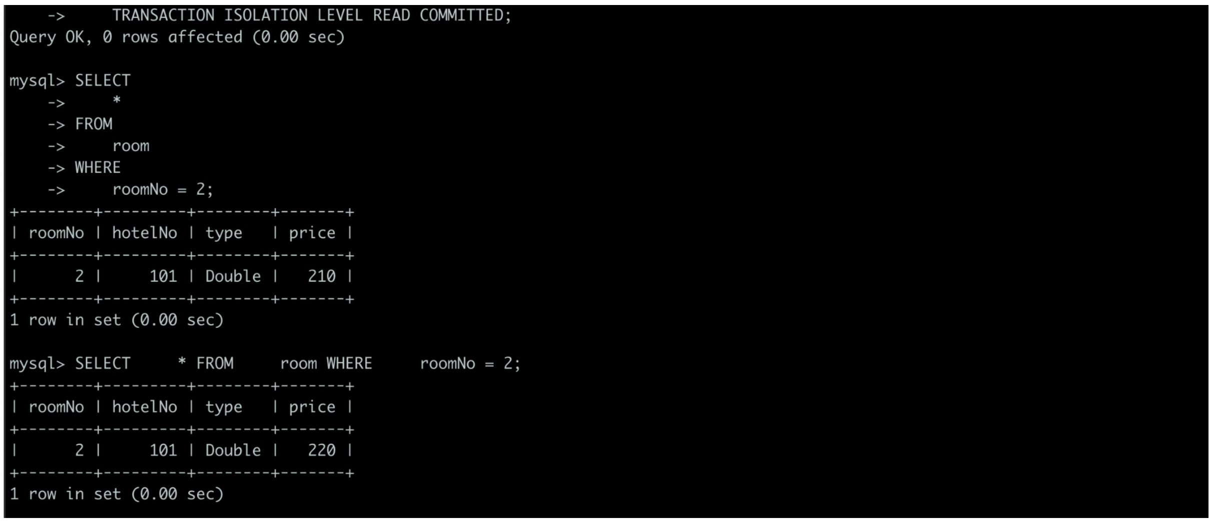


```
mysql> SET
    ->      TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT
    ->      *
    -> FROM
    ->      room
    -> WHERE
    ->      roomNo = 2;
+--------+---------+--------+-------+
| roomNo | hotelNo | type   | price |
+--------+---------+--------+-------+
|      2 |     101 | Double |   220 |
+--------+---------+--------+-------+
1 row in set (0.00 sec)
```

Read-committed:

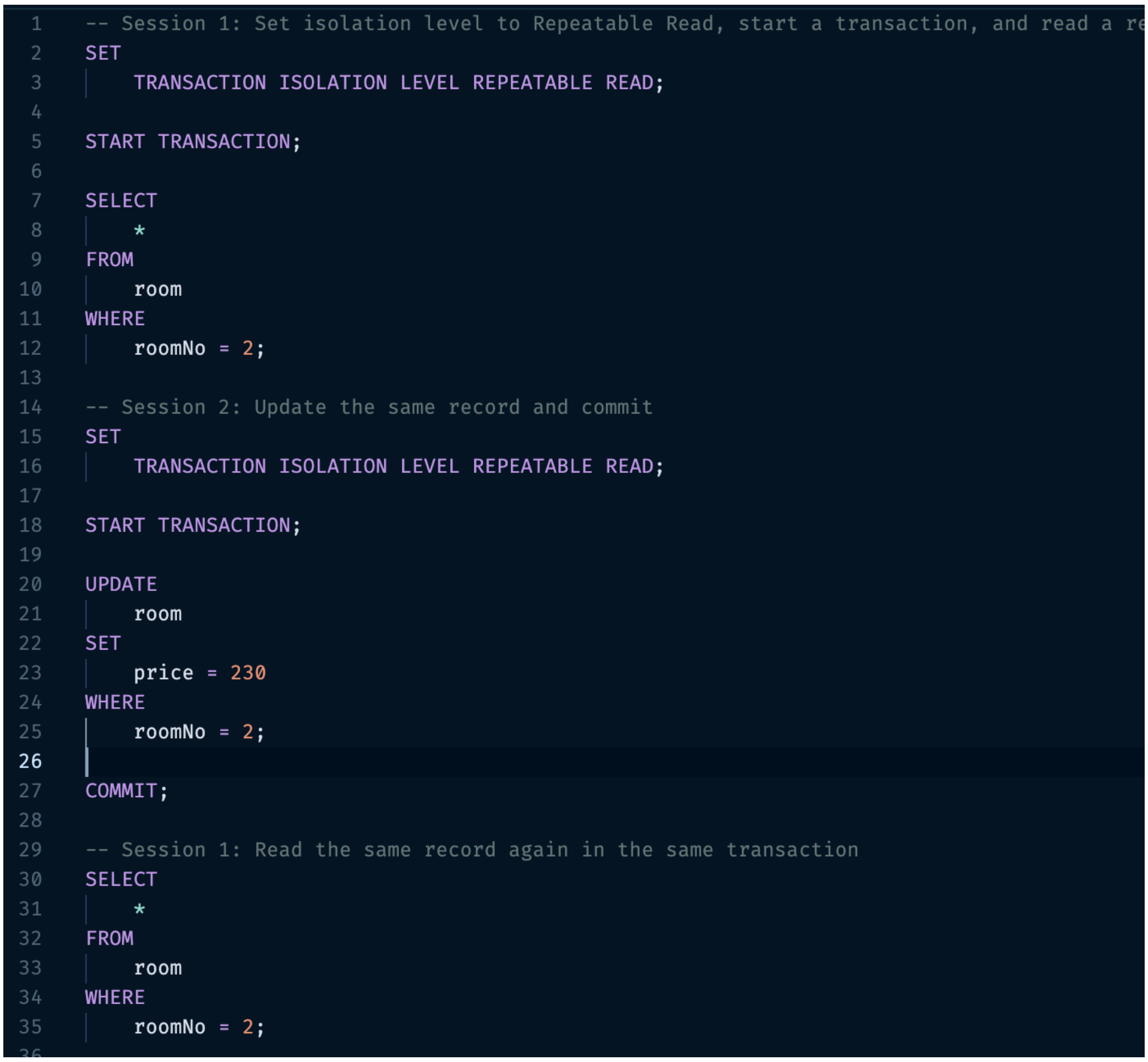


```
-- Session 1: Start a transaction and update a record without committing
START TRANSACTION;

UPDATE
    room
SET
    price = 220
WHERE
    roomNo = 2;

-- Session 2: Read the same record before the transaction in Session 1 is committed
SET
    TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT
    *
FROM
    room
WHERE
    roomNo = 2;

/* Expected Output: Room price shows 210, preventing dirty read */
-- Session 1: Commit the transaction
COMMIT;

-- Session 2: Read the record after the transaction in Session 1 is committed
SELECT
    *
FROM
    room
WHERE
    roomNo = 2;

/* Expected Output: Room price shows 220 */
```

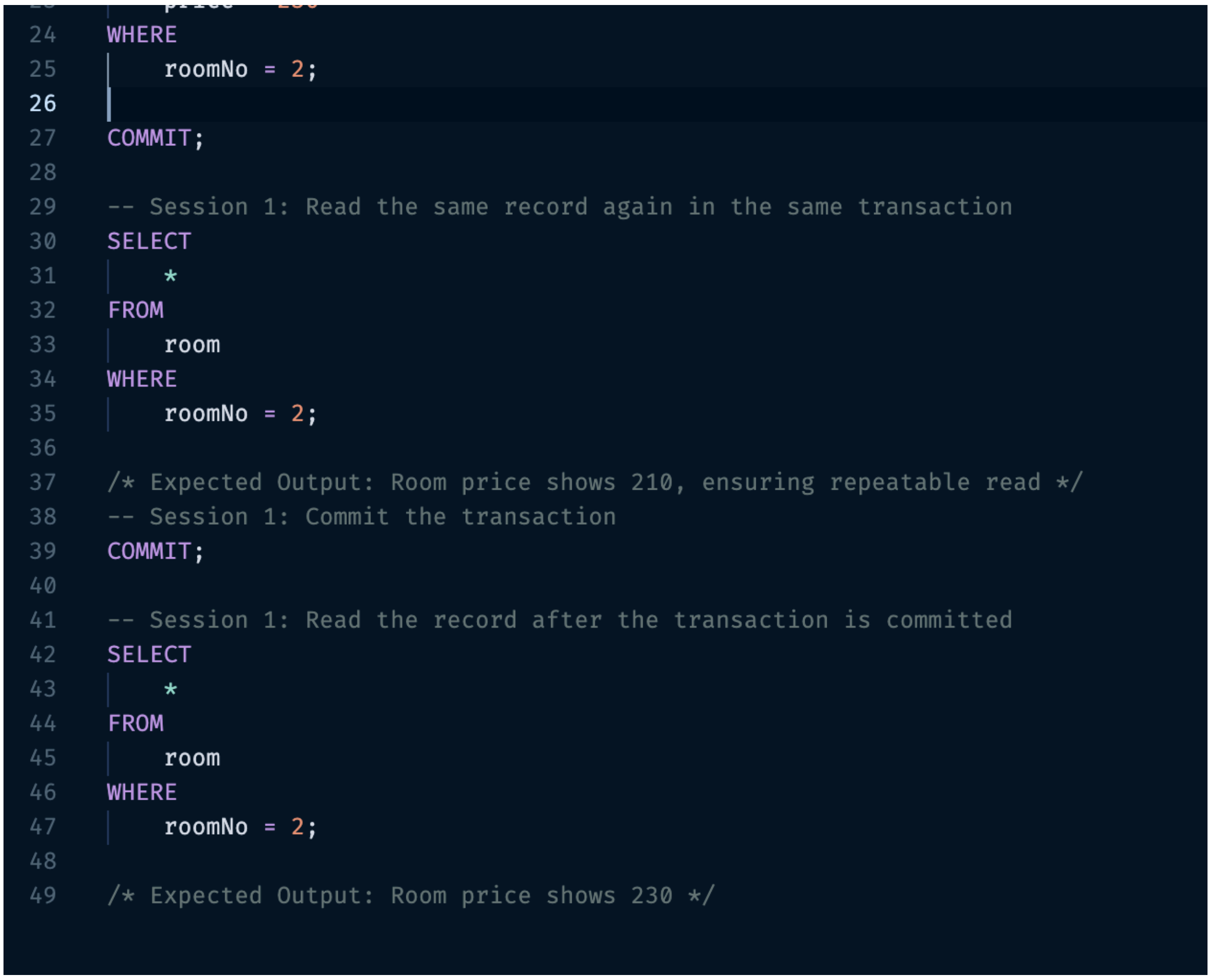


```
    ->      TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT
    ->      *
    -> FROM
    ->      room
    -> WHERE
    ->      roomNo = 2;
+--------+---------+--------+-------+
| roomNo | hotelNo | type   | price |
+--------+---------+--------+-------+
|      2 |     101 | Double |   210 |
+--------+---------+--------+-------+
1 row in set (0.00 sec)

mysql> SELECT      * FROM      room WHERE      roomNo = 2;
+--------+---------+--------+-------+
| roomNo | hotelNo | type   | price |
+--------+---------+--------+-------+
|      2 |     101 | Double |   220 |
+--------+---------+--------+-------+
1 row in set (0.00 sec)
```

Readable-read:

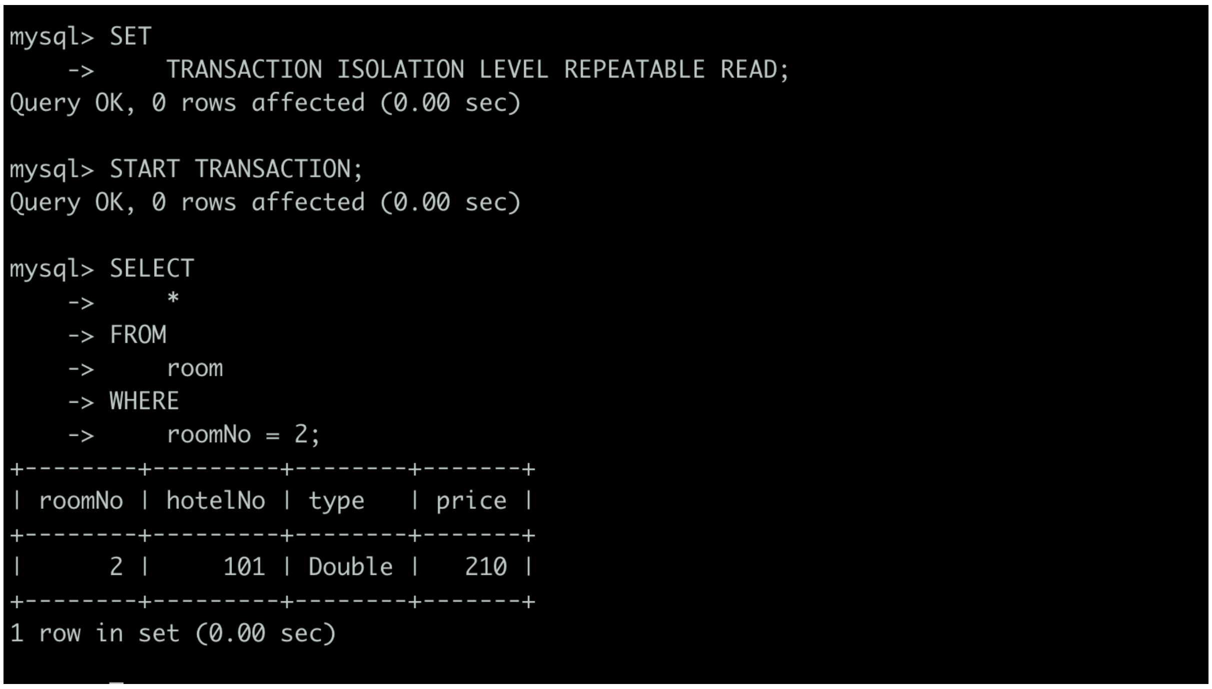


```sql
-- Session 1: Set isolation level to Repeatable Read, start a transaction, and read a re
SET
    TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

SELECT
    *
FROM
    room
WHERE
    roomNo = 2;

-- Session 2: Update the same record and commit
SET
    TRANSACTION ISOLATION LEVEL REPEATABLE READ;

START TRANSACTION;

UPDATE
    room
SET
    price = 230
WHERE
    roomNo = 2;

COMMIT;

-- Session 1: Read the same record again in the same transaction
SELECT
    *
FROM
    room
WHERE
    roomNo = 2;
```



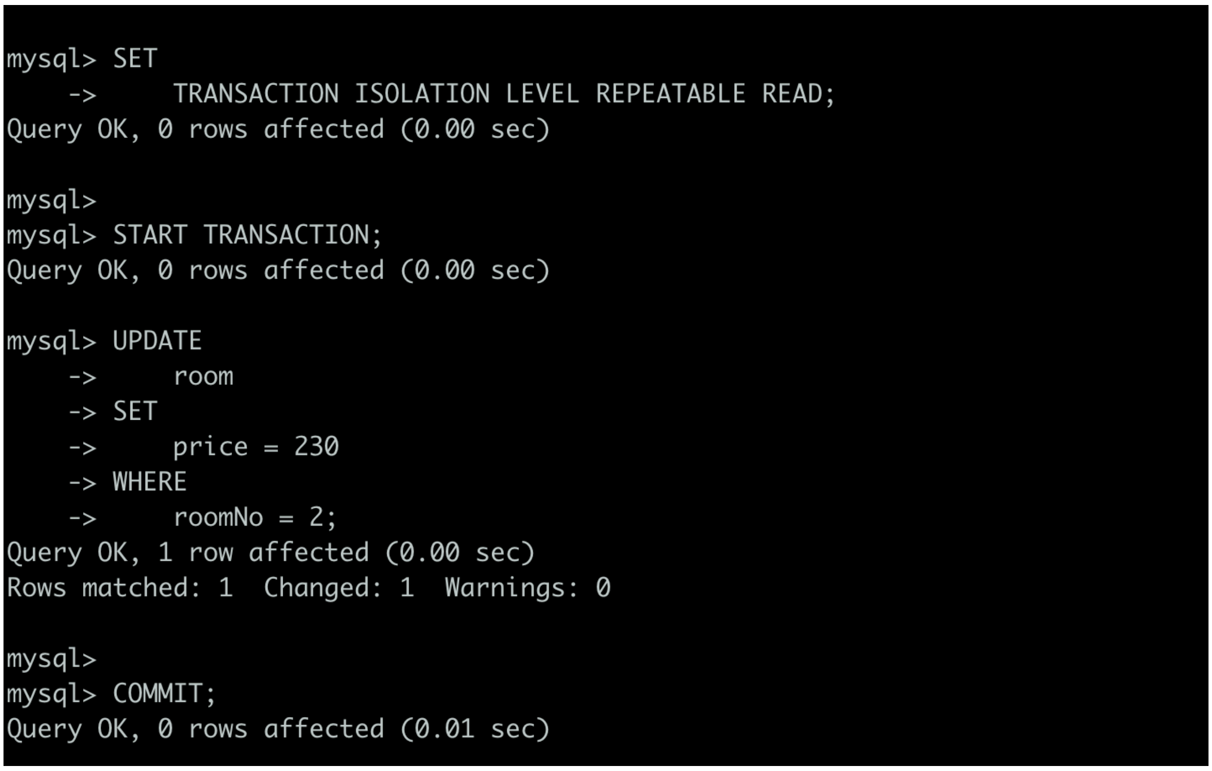
```sql
WHERE
    roomNo = 2;

COMMIT;

-- Session 1: Read the same record again in the same transaction
SELECT
    *
FROM
    room
WHERE
    roomNo = 2;

/* Expected Output: Room price shows 210, ensuring repeatable read */
-- Session 1: Commit the transaction
COMMIT;

-- Session 1: Read the record after the transaction is committed
SELECT
    *
FROM
    room
WHERE
    roomNo = 2;

/* Expected Output: Room price shows 230 */
```

Session 1



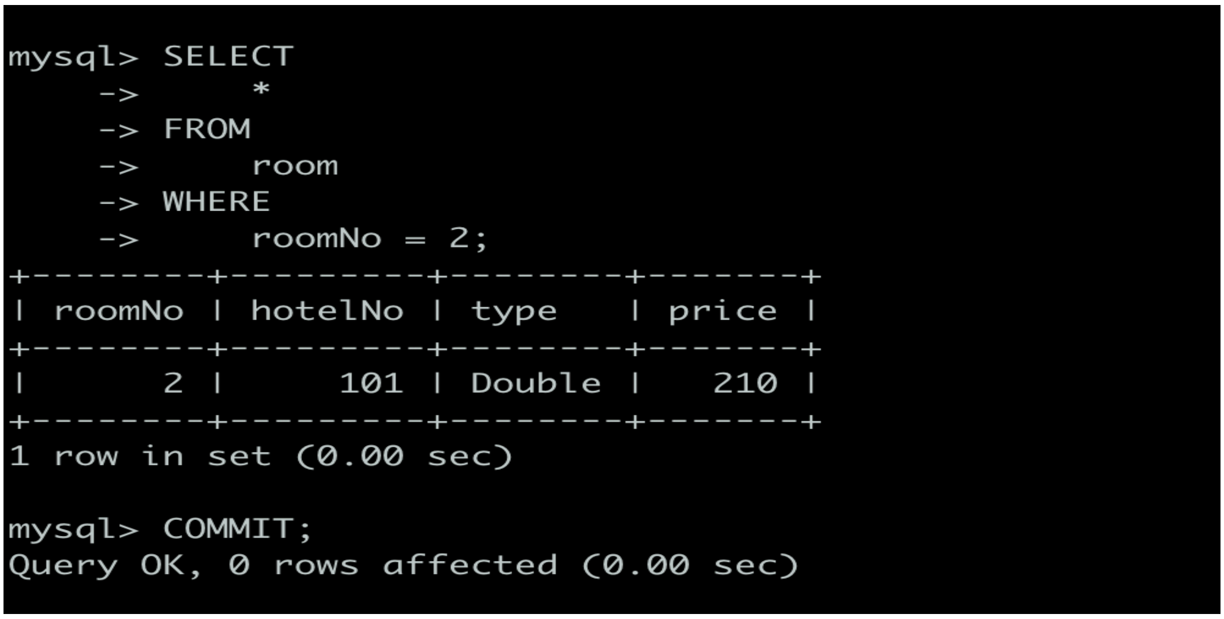
```
mysql> SET
    ->      TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.00 sec)

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

mysql> SELECT
    ->      *
    -> FROM
    ->      room
    -> WHERE
    ->      roomNo = 2;
+--------+---------+--------+-------+
| roomNo | hotelNo | type   | price |
+--------+---------+--------+-------+
|      2 |     101 | Double |   210 |
+--------+---------+--------+-------+
1 row in set (0.00 sec)
```

Session 2



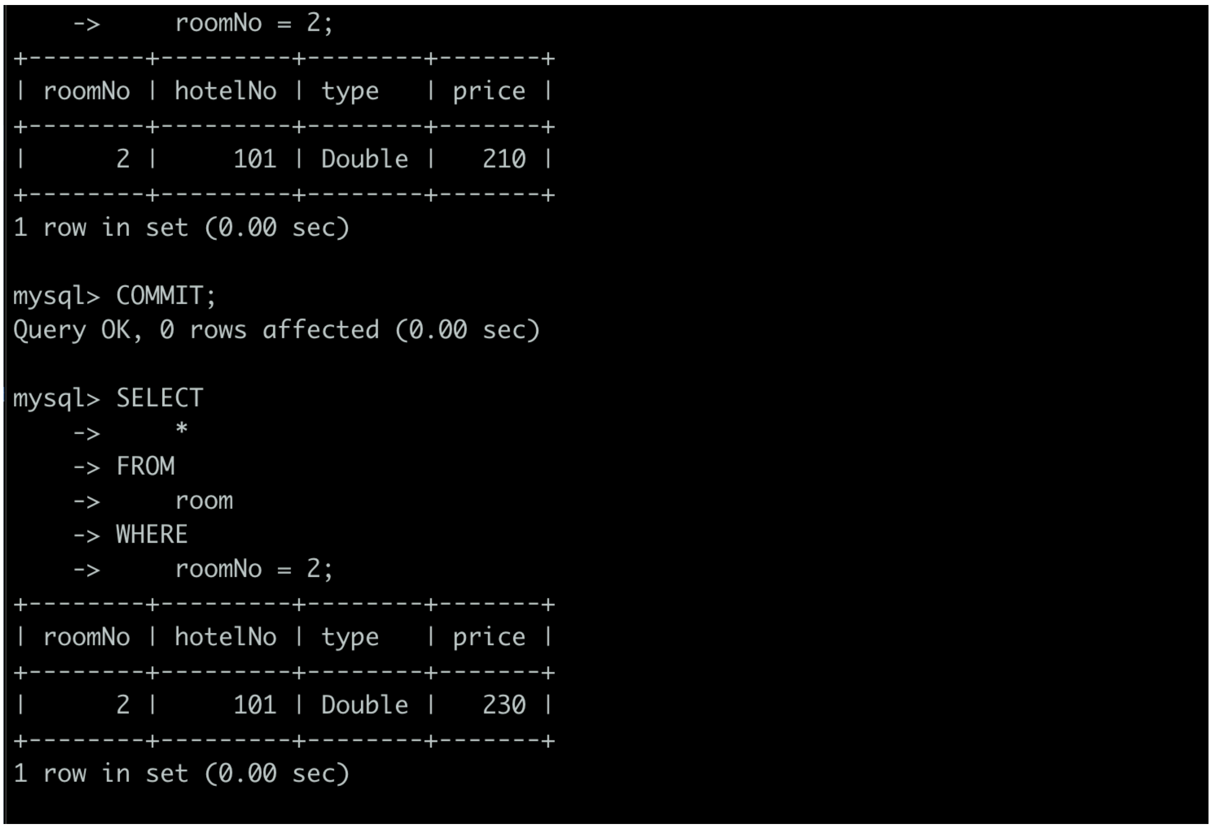
```
mysql> SET
    ->     TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.00 sec)

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

mysql> UPDATE
    ->     room
    -> SET
    ->     price = 230
    -> WHERE
    ->     roomNo = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

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

Session 1



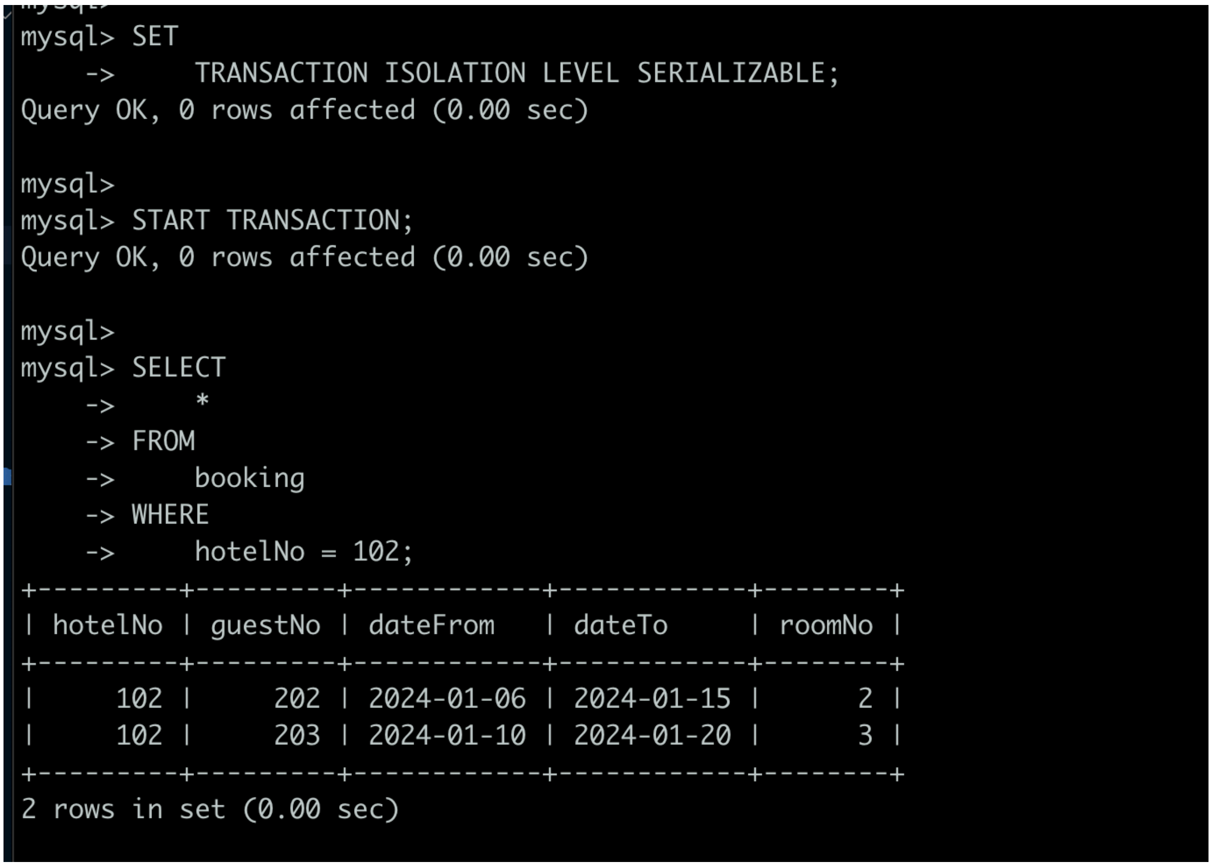
```
mysql> SELECT
    ->     *
    -> FROM
    ->     room
    -> WHERE
    ->     roomNo = 2;
+--------+---------+--------+-------+
| roomNo | hotelNo | type   | price |
+--------+---------+--------+-------+
|      2 |     101 | Double |   210 |
+--------+---------+--------+-------+
1 row in set (0.00 sec)

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

Session 1



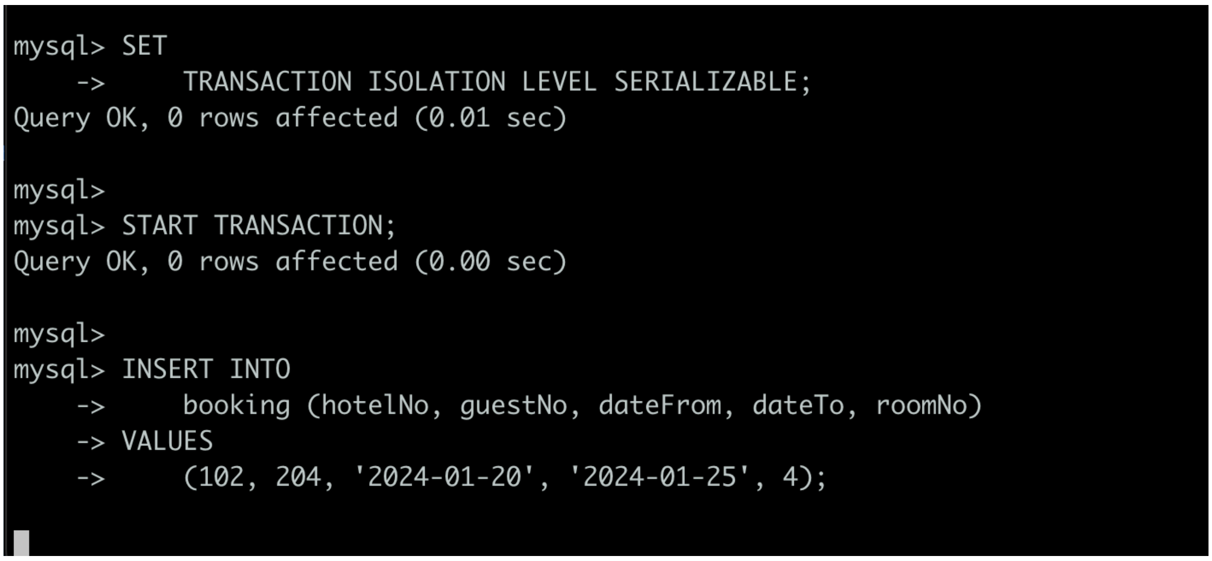
```
    ->     roomNo = 2;
+--------+---------+--------+-------+
| roomNo | hotelNo | type   | price |
+--------+---------+--------+-------+
|      2 |     101 | Double |   210 |
+--------+---------+--------+-------+
1 row in set (0.00 sec)

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

mysql> SELECT
    ->     *
    -> FROM
    ->     room
    -> WHERE
    ->     roomNo = 2;
+--------+---------+--------+-------+
| roomNo | hotelNo | type   | price |
+--------+---------+--------+-------+
|      2 |     101 | Double |   230 |
+--------+---------+--------+-------+
1 row in set (0.00 sec)
```

Serializable
Session 1



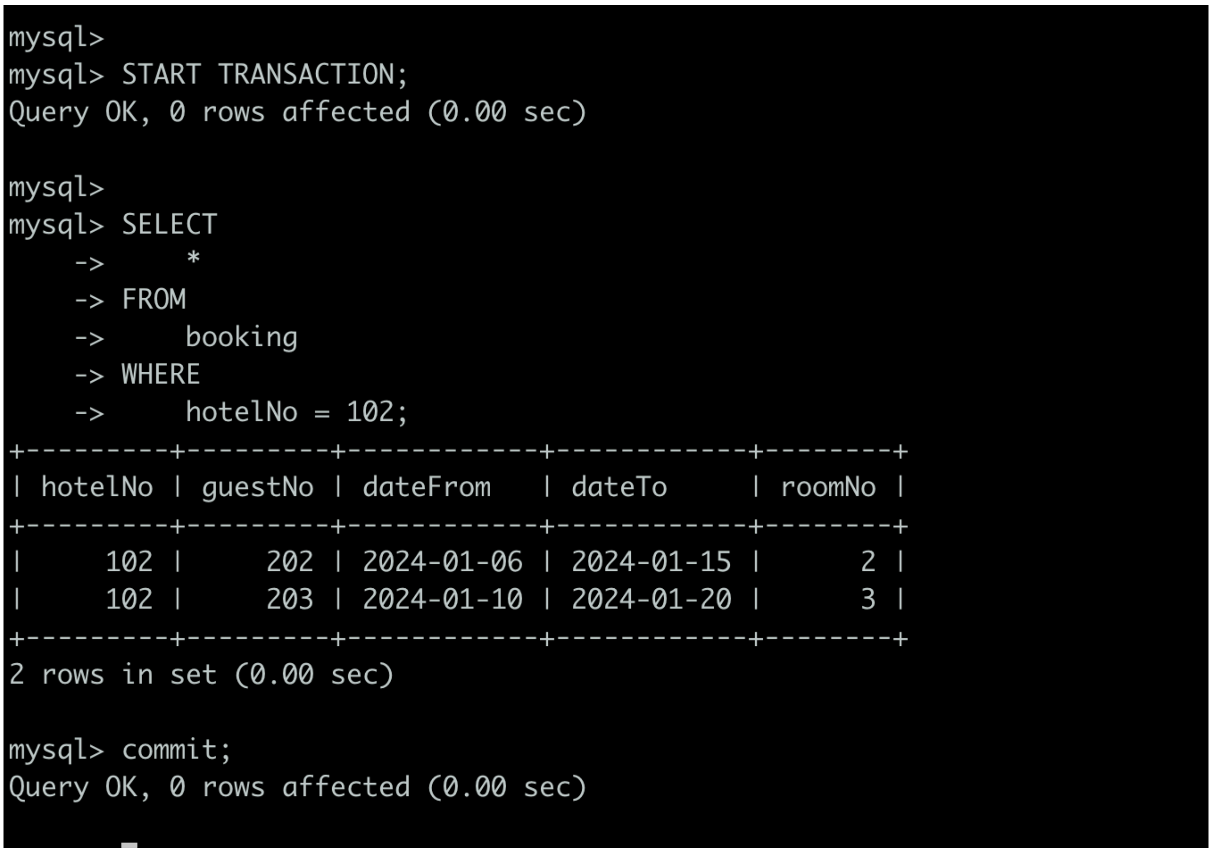
```
mysql> SET
    ->     TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.00 sec)

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

mysql>
mysql> SELECT
    ->     *
    -> FROM
    ->     booking
    -> WHERE
    ->     hotelNo = 102;
+---------+---------+------------+------------+--------+
| hotelNo | guestNo | dateFrom   | dateTo     | roomNo |
+---------+---------+------------+------------+--------+
|     102 |     202 | 2024-01-06 | 2024-01-15 |      2 |
|     102 |     203 | 2024-01-10 | 2024-01-20 |      3 |
+---------+---------+------------+------------+--------+
2 rows in set (0.00 sec)
```

Session 2 (Blocked unless Session 1 is committed)



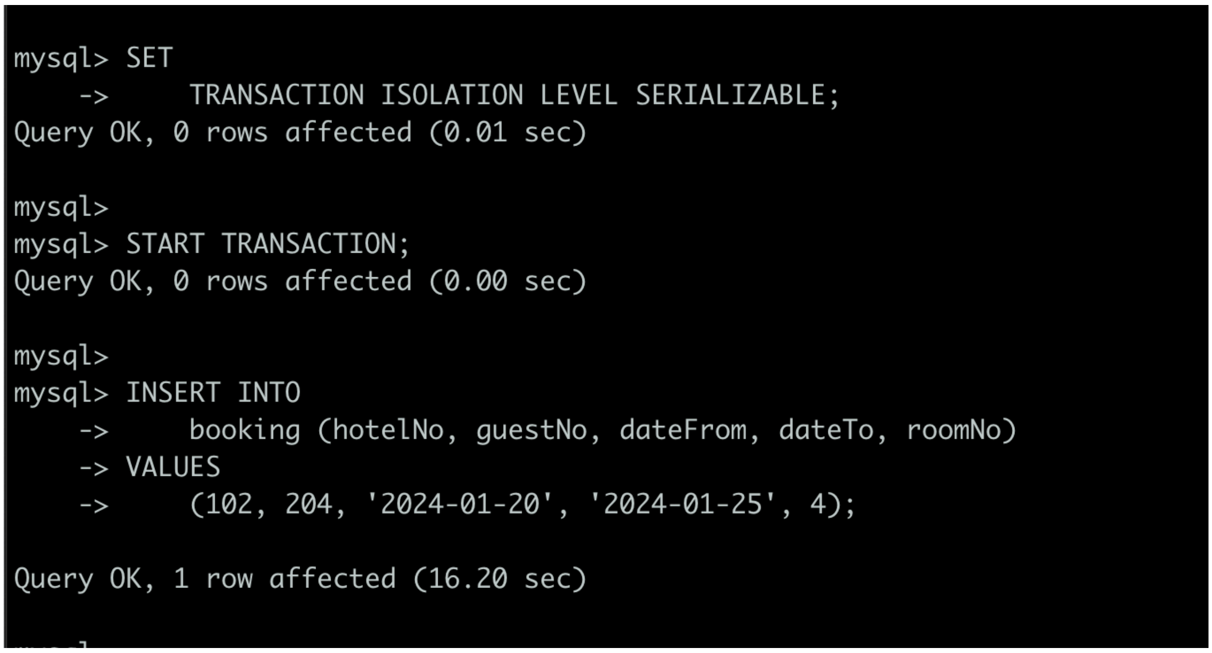
```
mysql> SET
    ->     TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.01 sec)

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

mysql>
mysql> INSERT INTO
    ->     booking (hotelNo, guestNo, dateFrom, dateTo, roomNo)
    -> VALUES
    ->     (102, 204, '2024-01-20', '2024-01-25', 4);
```

Session 1



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

mysql>
mysql> SELECT
    ->     *
    -> FROM
    ->     booking
    -> WHERE
    ->     hotelNo = 102;
+---------+---------+------------+------------+--------+
| hotelNo | guestNo | dateFrom   | dateTo     | roomNo |
+---------+---------+------------+------------+--------+
|     102 |     202 | 2024-01-06 | 2024-01-15 |      2 |
|     102 |     203 | 2024-01-10 | 2024-01-20 |      3 |
+---------+---------+------------+------------+--------+
2 rows in set (0.00 sec)

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

Session 2



```
mysql> SET
    ->     TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.01 sec)

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

mysql>
mysql> INSERT INTO
    ->     booking (hotelNo, guestNo, dateFrom, dateTo, roomNo)
    -> VALUES
    ->     (102, 204, '2024-01-20', '2024-01-25', 4);

Query OK, 1 row affected (16.20 sec)
```

4. Impact of Isolation Levels on Database Performance

Locking
Locking is a common mechanism used by database management systems (DBMS) to enforce isolation levels. Locks can be applied at various granularities, such as rows, pages, or tables, and can be of different types, including shared locks for read operations and exclusive locks for write operations.

Row Versioning
Row versioning, also known as Multiversion Concurrency Control (MVCC), allows multiple versions of a row to exist concurrently. Each transaction sees a consistent snapshot of the database at a specific point in time, which helps in reducing contention.

Impact on Transaction Throughput and System Overhead

Read Uncommitted: Minimal locking or versioning, resulting in the highest transaction throughput and lowest system overhead. Suitable for scenarios where data accuracy is less critical.

Read Committed: Uses locks to prevent dirty reads, leading to moderate overhead. Suitable for environments with a mix of read and write operations where non-repeatable reads are acceptable.

Repeatable Read: Requires more extensive locking or versioning to prevent non-repeatable reads, resulting in higher overhead. Suitable for systems where consistency during read operations is critical.

Serializable: Involves the most extensive locking or versioning to prevent all anomalies, resulting in the lowest transaction throughput and highest system overhead. Suitable for scenarios requiring the highest level of data consistency and integrity.

5. Case Studies

Scenario: Online Banking System

In the finance industry, data integrity and consistency are paramount due to the sensitive nature of financial transactions. Consider an online banking system where customers perform transactions such as transferring money between accounts, paying bills, and checking account balances. The requirements for transaction isolation levels in this context are driven by the need to ensure accurate and reliable financial records.

Typical Requirements

High Consistency	Financial transactions must be accurate and consistent. Even a small error can lead to significant financial loss and legal

	implications.
Data Integrity	Ensuring that all transactions are correctly processed and that account balances are accurate at all times.

Isolation Level: Serializable

Example	When a customer transfers $100 from their savings account to their checking account, the transaction must ensure that the amount is debited from the savings account and credited to the checking account accurately. The system must prevent other transactions from reading intermediate states where the transfer is partially complete.
Reason	Serializable isolation level ensures full isolation from other transactions, preventing anomalies like dirty reads, non-repeatable reads, and phantom reads. This is crucial in financial systems to maintain the integrity and consistency of financial data

Scenario: Electronic Health Record (EHR) System

In the healthcare industry, managing patient data securely and accurately is critical. An EHR system stores and manages patient health information, including medical history, test results, and treatment plans. The requirements for transaction isolation levels are influenced by the need for data accuracy, confidentiality, and compliance with regulations such as HIPAA.

Typical Requirements

Data Accuracy	Patient records must be up-to-date and accurate to ensure proper diagnosis and treatment.
Confidentiality	Patient data must be protected against unauthorized access and

	modifications.
Compliance	The system must comply with healthcare regulations regarding data handling and patient privacy.

Isolation Level: Repeatable Read

Example	When a doctor updates a patient's treatment plan, the transaction must ensure that the plan remains consistent throughout the session. If the doctor checks the treatment plan again during the same session, they should see the same information without any modifications by other transactions.
Reason	Repeatable Read isolation level ensures that if a transaction reads a record multiple times, it will get the same values each time, preventing non-repeatable reads. This is important in healthcare to maintain consistent and accurate patient records while still allowing for some level of concurrent access to the database.

6. Conclusion

In this paper, we explored the fundamental concepts of database transactions and the significance of isolation levels in maintaining data integrity and consistency in multi-user environments. We began with a clear definition of database transactions and emphasized their critical role in scenarios such as banking systems and e-commerce platforms. We then delved into the ACID properties—Atomicity, Consistency, Isolation, and Durability—that underpin reliable transaction processing.

Key Points Recap

Definition and Importance: Database transactions are essential for ensuring data integrity and consistency, particularly in multi-user environments where concurrent data access is common.

ACID Properties:

Atomicity	Ensures transactions are all-or-nothing.
Consistency	Maintains database rules and integrity.

Isolation	Manages concurrent transactions to ensure accurate computations.
Durability	Guarantees that once committed, changes are permanent.

Isolation Phenomena:

Dirty Reads	Reading uncommitted changes
Non-Repeatable Reads	Different results for repeated reads of the same data.
Phantom Reads	Different sets of rows for repeated queries.
Serialization Anomalie	Discrepancies between concurrent and serial transaction execution.

Isolation Levels:

Read Uncommitted	High performance with potential data inconsistencies.
Read Committed	Prevents dirty reads.
Repeatable Read	Prevents non-repeatable reads
Serializable	Prevents all anomalies, ensuring full isolation.

Performance and Trade-offs:

Locking and row versioning impact transaction throughput and system overhead.
No single isolation level fits all scenarios due to varying performance and consistency needs.

Practical Importance
Understanding and choosing the appropriate isolation level is crucial for designing robust and efficient database systems. The choice of isolation level impacts the balance between data consistency and system performance. For instance, finance systems require the highest level of consistency and thus often use Serializable isolation. In contrast, applications like logging systems can tolerate some inconsistencies for better performance, making Read Uncommitted a viable choice.

By carefully considering the specific needs of an application, database designers can select the isolation level that provides the optimal balance of performance and data integrity. This knowledge is essential for developing systems that are both efficient and reliable, meeting the demands of various industries from finance to healthcare.

In conclusion, a thorough understanding of database transactions and isolation levels is indispensable for creating systems that uphold data integrity, manage concurrent access effectively, and perform efficiently under diverse operational conditions.