Database Transactions

Introduction

A database transaction is a series of actions carried out as one unit that guarantees data consistency and integrity, where numerous users are accessing and changing the data at once. It is a procedure that either finishes completely or doesn't occur at all. For Example: If we want to move money from one account to another in a bank, this requires taking money out of one account and adding it into another. To preserve data integrity, both of these procedures must either succeed or fail completely.

In multi user contexts, transactions are essential for maintaining data consistency and integrity. This is especially true for systems like e-commerce platforms and financial systems. The following situations are mentioned to underline the significance of transactions:

Banking Systems

Moving Funds Between Accounts

For instance, Bisho would like to move $500 to Gaurav's account from her own. Gaurav's account is credited and Bisho's account is debited during the process. These actions have to be atomic, which means that either both or neither can occur. The transaction has to make sure that neither account shows an incomplete state in the event that a system crash happens after Bisho's account is debited but before Gaurav's account is credited. Data integrity is maintained by transactions, which guarantee that either both changes are applied or none are.

ATM Extraction

Let’s assume Gaurav takes out $100 from an ATM. The system has to make sure that Gaurav's account balance shows the withdrawal amount correctly. Gaurav might get more money than he should have if a system fault happens after the cash is paid out but before the account balance is updated. Transactions make sure that both operations are carried out effectively or are rolled back, which guarantees data consistency.

E-commerce Platforms

A client purchases a laptop. Reducing the inventory count, handling the payment, and creating an order record are the several procedures involved in this. To make sure the inventory count is accurate, the transaction needs to be rolled back if the inventory count is decreased but the payment is unsuccessful. Transactions ensure consistency by guaranteeing that either every step is completed successfully or none at all.

Managing Shopping Carts

A customer puts things in their cart and checks out. Both the items and the user's money must be accepted during checkout. The transaction must make sure that the user is notified and that the cart is updated if an item becomes unavailable during the process. Transactions ensure that users don't buy things that are out of stock by helping to control concurrent access to inventory.

Properties of Transactions:

Atomicity:

It Guarantees that every activity in a transaction is finished completely or not at all. The transaction is rolled back in its entirety and the database is not updated if any portion of it fails. In the event that a transaction fails, all modifications performed up to that point are reversed, returning the database to its initial state prior to the transaction. A transaction is considered committed when all of its actions have been successfully finished, rendering any modifications irreversible.

Bank Transfers:

Debiting Bisho's account and crediting Gaurav's account are the two steps involved in transferring funds from Bisho to Gaurav.
Here, maintaining accurate account balances requires the proper completion of both activities. Data corruption would result if the debit operation is successful but the credit operation is unsuccessful (for example, because of a network problem). In this scenario, Bisho's account would be debited without Gaurav's account being credited. If any part of the transaction fails, the system rolls back the debit from Bisho's account, ensuring neither account is incorrectly updated.

Order processing scenario for e-commerce:

Reducing the amount of inventory, billing the customer's payment method, and generating an order record are the several processes involved in processing an order.
The system has to return the inventory count to its initial value in the event that the inventory reduction is finished but the payment processing is unsuccessful. Sales loss and erroneous inventory levels could result from not doing this. If any stage in the order processing transaction fails, the entire transaction is rolled back to prevent any partial modifications.

Consistency:

Transactions contribute to the consistency and integrity of databases by enforcing atomicity which means making sure that either every transaction's operations are finished or none at all. Partial updates are avoided since the transaction rolls back if any operation fails. Transactions help in respecting Integrity Constraints by taking care to adhere to regulations about primary keys, foreign keys, unique constraints, and non-null constraints. Regressions lead to a rollback.

Isolation:

When used in conjunction with other concurrent transactions, iolation guarantees that each one runs separately and without hindrance from the others. This avoids problems that can result in erroneous data and computation mistakes, such as unclean reads, non-repeatable reads, and phantom reads.
Importance of Isolation:
Eliminates Conflicts in Data: guarantees that no transaction reads or writes intermediate data from another running transaction.
Preserves Uniformity: ensures that simultaneous operations yield identical outcomes to those obtained from sequential execution.
Prevents Abnormalities: Avoids anomalies such as non-repeatable reads (data changing between reads), phantom reads (new data appearing during transaction), and dirty reads (reading uncommitted data).

Durability

Durability makes ensuring that, even in the case of system failures, a transaction's modifications are eternally saved in the database after it is committed.
Write-Ahead Logging: Before a change is implemented to the database, it is first recorded in a log. The log is used to recover committed transactions in the event of a crash.
Database checkpoints: To guarantee that changes are securely recorded, periodic snapshots of the database's current state are taken.
Systems for backup and recovery: After a failure, the database can be consistently restored with the aid of reliable recovery techniques and routine backups.
Example Order Processing: The order information are securely saved when the order transaction is committed. The order can still be properly collected and processed from the logs in the event that the system breaks.
Durability ensures that committed modifications remain enduring and resistant to errors, guaranteeing dependable database operations.

Isolation Levels

Introduction

Data consistency in concurrent situations is ensured via isolation in database transactions. According to the SQL standard, four phenomena are forbidden at different levels of isolation:
Reading uncommitted modifications from another transaction is known as a "dirty read." For instance, if Transaction B views a record that Transaction A has modified before A commits, B will see incorrect data in the event that A rolls back.
Non-Repeatable Reads: Reading the same data twice and receiving different results because of changes made by a previous committed transaction. When Transaction A reads a record, for example, and Transaction B modifies and commits it, and then Transaction A reads the record again, A observes the change.

Phantom Reads: Reading a collection of rows that have been updated by insertions or deletions made in a previous committed transaction. As an illustration, suppose Transaction A reads rows that satisfy a requirement, Transaction B inserts a new row that satisfies that criteria, and then A reads the new row once more.
Serialization anomalies: Transactions carried out concurrently produce different results than those carried out sequentially. In contrast to serial execution, the outcome may be inconsistent if two transactions move money between accounts at the same time.

Overview of Isolation Levels

Read Uncommitted

Changes made in one transaction are visible to all other transactions prior to their commit when they are in the read uncommitted isolation level. This could result in a few situations:
Dirty Reads: A record is updated by Transaction A. This commit is read by transaction B. Transaction B has read erroneous data if Transaction A rolls back.
Interim Data Visibility: Records are added or removed by Transaction A. Even though these modifications may not be permanent, Transaction B sees them right away, which could cause discrepancies if Transaction A reverses course.
Inconsistent Analysis: Transaction A makes several modifications. An inconsistent snapshot of the database state is produced when Transaction B reads the data in the middle of an update. For Example: Transaction A debits one account to begin the transfer of monies. The uncommitted new balance is read by Transaction B. B has utilized false information if Transaction A fails.

Read Committed

A transaction can only read data that has been committed by other transactions when it is in the Read Committed isolation level. Non-repeatable readings are not prevented by this, but unclean reads are. Until a transaction is committed, it cannot read data that is presently being altered by another transaction. This guarantees that all read data is reliable and validated.
Non-Repeatable Reads Allowed: If a transaction reads a record and then changes it and commits the modifications, the first transaction will read the record again and see the updated value. This implies that if a query is run more than once within a single transaction, it may yield different results each time.
For instance, a bank balance check
Step 1: Transaction A obtains an account's balance ($1000).
Step 2: After updating the amount to $900, Transaction B commits.
Step 3: Because the data has changed between reads, if Transaction A checks the balance once more, it finds $900, indicating a non-repeatable read.

Repeatable Read

A transaction that executes at the Repeatable Read isolation level will never encounter a non-repeatable read since it will always see the same values for each record it reads more than

once. Nevertheless, additional rows updated by other transactions may still be seen if the transaction requests the database again because this level does not prohibit phantom reads.
Preventing Non-Repeatable Reads: Regardless of whether other transactions make changes to the record in between, once a transaction reads it, it will always return the same value on subsequent reads.
Enabling Phantom Reads: When a transaction does a re-query on a range of records (e.g., all rows satisfying a particular criteria), any additional rows that are inserted or deleted within that range will be visible to the original transaction.
Step 1: A product's amount (10 units) is read by Transaction A.
Step 2: After updating the quantity to five units, Transaction B commits.
Step 3: Despite preventing non-repeatable reads, Transaction A reads the same product quantity again and continues to observe 10 units.

Serializable

The ultimate degree of isolation, known as Serializable, guarantees complete separation from other transactions. All abnormalities are avoided, including ghost readings, unclean reads, and non-repeatable reads. Rather of being executed concurrently, transactions seem to be executed one after the other in a sequential fashion. It prevents other transactions from changing or adding rows that could alter the data of the current transaction, ensuring data consistency.
For Example: Bank Transfers and Balance Checks: In order to ensure that there is no interference, Transaction B must wait until Transaction A is finished sending funds before proceeding with the balance check or record modification.

Comparison of Isolation Levels

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

Practical Demonstration

Read Uncommitted:

Allows dirty reads. Transaction 2 reads the uncommitted change from Transaction 1.

```
mysql> -- Set isolation level to Read Uncommitted
Query OK, 0 rows affected (0.00 sec)

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

mysql>
mysql> -- Transaction 1: Start transaction and update stock price
Query OK, 0 rows affected (0.00 sec)

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

mysql> UPDATE stocks SET price = 155.00 WHERE id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql>
```

```
mysql> -- Transaction 2: Start another transaction and read stock price
Query OK, 0 rows affected (0.00 sec)

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

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 155.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 155.00 (dirty read), Actual: 155.00
Query OK, 0 rows affected (0.00 sec)
```

Read Committed:

Prevents dirty reads. Transaction 2 can only read committed data, so it sees the original price until Transaction 1 commits.

```
mysql> -- READ COMMITTED
Query OK, 0 rows affected (0.00 sec)

mysql> -- Set isolation level to Read Committed
Query OK, 0 rows affected (0.00 sec)

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

mysql>
mysql> -- Transaction 1: Start transaction and update stock price
Query OK, 0 rows affected (0.00 sec)

mysql> -- Reset price to 150 before transaction
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE stocks SET price = 150.00 WHERE id = 1;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0

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

mysql> UPDATE stocks SET price = 155.00 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

BEFORE COMMITING TRANSACTION T1:
Expected: 150, Actual: 150

```
mysql> -- Transaction 2: Start another transaction and read stock price
Query OK, 0 rows affected (0.01 sec)

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

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 150.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 150.00 (no dirty read), Actual: 150.00
Query OK, 0 rows affected (0.00 sec)
```

AFTER COMMITING THE TRANSACTION T1:
Expected: 155, Actual: 155

```
mysql> -- Transaction 2: Read stock price again
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 155.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 155.00, Actual: 155.00
Query OK, 0 rows affected (0.00 sec)
```

REPEATABLE READ:

Prevents non-repeatable reads. Transaction 1 sees the same data for the duration of its execution, even if Transaction 2 commits changes.

Terminal 1:

```
mysql>
mysql> -- REPEATABLE READ
Query OK, 0 rows affected (0.00 sec)

mysql> -- Set isolation level to Repeatable Read
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
ERROR 1568 (25001): Transaction characteristics can't be changed while a transac
tion is in progress
mysql>
mysql> -- Reset price to 150 before transaction
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE stocks SET price = 150.00 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> -- Transaction 1: Start transaction and read stock price
Query OK, 0 rows affected (0.00 sec)

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

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 150.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 150.00, Actual: 150.00
Query OK, 0 rows affected (0.00 sec)

mysql> -- Transaction 1: Read stock price again
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 150.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 150.00 (repeatable read), Actual: 150.00
Query OK, 0 rows affected (0.00 sec)

mysql> -- Commit Transaction 1
Query OK, 0 rows affected (0.00 sec)

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

Terminal 2:

```
mysql> -- Transaction 2: Start another transaction and update stock price
Query OK, 0 rows affected (0.00 sec)

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

mysql> UPDATE stocks SET price = 155.00 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> -- Commit Transaction 2
Query OK, 0 rows affected (0.00 sec)

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

mysql>
mysql> -- Check final state of the stock price
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 155.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 155.00, Actual: 155.00
Query OK, 0 rows affected (0.01 sec)
```

Serializable Read:

The strictest isolation level, ensuring complete isolation. Transaction 2 cannot update until Transaction 1 completes, preventing any concurrent updates that would affect the data.

Terminal 1:

```
mysql> -- SERIALIZABLE READ
Query OK, 0 rows affected (0.00 sec)

mysql> -- Set isolation level to Serializable
Query OK, 0 rows affected (0.00 sec)

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

mysql>
mysql> -- Reset price to 150 before transaction
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE stocks SET price = 150.00 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> -- Transaction 1: Start transaction and read stock price
Query OK, 0 rows affected (0.00 sec)

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

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 150.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 150.00, Actual: 150.00
Query OK, 0 rows affected (0.00 sec)

mysql> -- Commit Transaction 1
Query OK, 0 rows affected (0.01 sec)

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

Terminal 2:

```
mysql> -- Transaction 2: Start another transaction and try to update stock pr
ice
Query OK, 0 rows affected (0.01 sec)

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

mysql> UPDATE stocks SET price = 155.00 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> -- Expected: Block until Transaction 1 is committed or rolled back
Query OK, 0 rows affected (0.00 sec)

mysql> -- Transaction 2: Now able to update stock price
Query OK, 0 rows affected (0.00 sec)

mysql> -- Commit Transaction 2
Query OK, 0 rows affected (0.00 sec)

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

mysql>
mysql> -- Check final state of the stock price
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT price FROM stocks WHERE id = 1;
+--------+
| price  |
+--------+
| 155.00 |
+--------+
1 row in set (0.00 sec)

mysql> -- Expected: 155.00, Actual: 155.00
Query OK, 0 rows affected (0.01 sec)
```

These scripts demonstrate how each isolation level affects the visibility of changes made by concurrent transactions on a stock market table.

Impact of Isolation Levels on Database Performance

Performance Analysis:

Database Management Systems achieve isolation through the use of locking and row versioning. While these procedures are essential for preserving data consistency, they can also affect system overhead and transaction throughput.
Transaction Throughput: Transactions may have to wait for locks to be released, which could cause delays. As a result, locking can lower throughput.
Deadlocks: Deadlock detection and resolution techniques are necessary when two or more transactions are waiting for one another to release locks.
Lock Contention: In high-concurrency systems in particular, a high level of lock contention can severely impair performance.

Row Versioning Process

Snapshot Isolation: This technique gives each transaction a "snapshot" of the database at a specific moment in time by using row versions. The previous and updated versions of a row are stored side by side when it is modified. Transactions can work on a snapshot of the database with Multiversion Concurrency Control enabled, without interfering with other transactions. Every transaction starts with a view of the data that is consistent throughout.
Effect on Capabilities:
Transaction Throughput: By lowering the requirement for locks and enabling more concurrent transactions, row versioning can increase throughput.
Storage Overhead: Keeping track of many data versions raises the need for storage and complicates garbage collection to get rid of outdated versions.
Read Performance: Because reads don't need to be locked and can start right away without waiting on other transactions, they can happen more quickly.
Comparison and Tradeoffs:

Mechanism	Benefits	Drawbacks
Locking	Simple to implement, strong consistency	Reduced throughput, potential for deadlocks, lock contention
Row Versioning	High concurrency, fewer locks, better read performance	Increased storage overhead, complexity in version management

TRADE-OFFS:

Since each isolation level includes trade-offs between resource usage, performance, and data consistency, there isn't one that works for every situation. Depending on the particular requirements of the application, different isolation degrees are best for various use scenarios.

READ UNCOMMITED

It is perfect for situations when high performance is necessary but data quality is less important. For instance, data warehousing or logging systems, when quick, approximative outputs are acceptable. It gives excellent performance and throughput because of little locking. There is a high chance of unclean reads, which could result in inconsistent data.

READ COMMITTED

It is fit for applications where performance and consistency must be balanced. An illustration of this would be online transaction processing (OLTP) platforms such as e-commerce sites, where data must be up to date but little discrepancies are acceptable. It is more consistent than Read Uncommitted, prevents unclean reads. It enables phantom and non-repeatable reads, which in some usage scenarios may result in inconsistent results.

REPEATABLE READ

It is fit for applications that require more consistency but yet have room for aberrations. Financial applications, such as creating monthly statements, are an example of where repeated reads inside a transaction should be consistent. It provides strong consistency for repeated reads is provided, preventing filthy and non-repeatable reads. Here, phantom reads are permitted, although range queries may still result in inconsistent results.

Serializable Application:

It is crucial for applications where performance can take a backseat and the highest level of consistency is demanded. In crucial financial transactions where precision and consistency are crucial, such as banking systems, it is a must. It guarantees the highest degree of data consistency by preventing any irregularities. It may drastically impair performance because of excessive locking and possible transaction blocking, which lowers throughput.

Isolation Level	Ideal Scenarios	Example Applications	Trade-offs
Read Uncommitted	High performance, low data accuracy	Logging systems, data warehousing	High risk of dirty reads, potential data inconsistencies

	needed		
Read Committed	Balance between consistency and performance	E-commerce platforms, general OLTP systems	Prevents dirty reads, but allows non-repeatable and phantom reads
Repeatable Read	High consistency for repeated reads	Financial applications, reporting systems	Prevents dirty and non-repeatable reads, but allows phantom reads
Serializable	Highest consistency, critical accuracy needed	Banking systems, critical financial transactions	Prevents all anomalies, but can reduce performance and throughput

Case Studies:

Case I: E-commerce

A variety of transactions are managed by e-commerce systems, such as product listings, order placements, inventory changes, and payment processing. In order to accommodate potentially massive volumes of concurrent users, these systems must retain high performance while guaranteeing data quality and consistency.

Important Scenarios in Order Placement:

The system needs to make sure the product is available at the time the customer places an order. A user's uncommitted transactions should not cause a price or quantity change.
For Example:
Step 1: After visiting the product page, Customer A discovers that there are ten units available.
Step 2: Two units are ordered by Customer A. To make sure the quantity is compared to the most recent committed state, the system locks the record at this point.
Step 3: When Customer B places an order at the same time as Customer A, they will see the inventory's committed state (eight units if Customer A's purchase is committed) and base their order on this updated information.

Needs for Inventory Management:

In order to prevent overselling, the system needs to make sure that the number of products does not go negative. This can occur when several transactions read the same stock level before it is updated. As soon as transactions are committed, inventory levels need to accurately reflect the data.
For Example:
Step 1: After receiving a shipment, an employee adds more stock to the inventory.
Step 2: In order to prevent customers from attempting to make purchases based on false stock information, this update must be committed before customers can view the updated stock levels.

Conditions for Processing Payments:

In order to prevent the customer from being charged in the event that the order placing fails, the entire payment transaction must either complete completely or not at all. It guarantees that the merchant's account is credited and the customer's account is accurately debited.
For Example:
Step1: Initiating payment for their order is Customer A.
Step 2: The payment is processed by the system, which credits the merchant's account and debits Customer A's account. Before any additional transactions can change the involved accounts, these processes must be carried out.

Balancing Consistency and Performance

Control of Concurrency:

Read Committed Level: By simply locking rows for brief intervals during updates, this level enables the system to efficiently handle a high number of concurrent transactions.
Possibility of Phantom and Non-Repeatable Reads: Although such anomalies are conceivable, in e-commerce, the advantages of improved performance typically outweigh the requirement for more stringent consistency standards. It makes sure the system keeps responding even when there is a lot of traffic, like during special promotions or holiday sales. It reduces the overhead of locking, enabling the database to handle a large number of requests without experiencing a noticeable drop in performance.
The Read Committed isolation level in e-commerce provides the best possible compromise between preserving data consistency and guaranteeing fast performance. It guarantees that users constantly interact with committed, dependable data by prohibiting dirty reads, all the while enabling the system to manage numerous transactions at once. Maintaining precise inventory levels, offering a flawless shopping experience, and guaranteeing dependable order and payment processing all depend on this balance.

Case II: Stock Market Transactions

Trades, orders, and revisions to market data are extremely delicate and urgent activities that take place in the stock market. To preserve the integrity of the market and investor confidence, these systems need to guarantee the highest level of accuracy and consistency while managing a large number of concurrent transactions.
Serializable:
It avoids all anomalies, such as ghost reads, unclean reads, and non-repeatable reads and guarantees complete data integrity and totally segregated processing of every transaction, both of which are necessary for precise trading and market operations.
Ensuring that buy and sell orders are matched accurately in the absence of interruption from other ongoing transactions is known as accurate order matching. To avoid problems like the same order being executed twice, transactions pertaining to the placement and execution of orders must be kept separate.
Workflow Example:
Step 1: A buy order is placed for a certain stock by Trader A.
Step 2: The order is matched with a matching sell order after being submitted into the order book.
Step 3: The order is executed correctly and reflected in the accounts of both traders when the transaction is committed.

Updates to Market Data Requirements:

Accuracy of real-time data guarantees that market data, including volumes and stock prices, are updated with precision in real-time by maintaining data consistency even in high trading volumes. This technique isolates transactions to preserve data consistency.
Example steps:
Step 1: A new trade is made, which changes the volume and price of the stock.
Step 2: In order to reflect the updated price and volume, the system updates the market data.
Step 3: The transaction is committed, guaranteeing that the most recent market data will be reflected in all ensuing reads.

Needs for Portfolio Management:

There needs to be a guarantee for continuous updates to portfolio appraisals according to trades and market fluctuations. In order to avoid inconsistent valuation, transactions involving portfolio updates must be isolated.
For Example:
Step 1: Based on recent trades, Investor A's portfolio is reevaluated.
Step 2: After all committed transactions are taken into account, the system determines the new portfolio value.
Step 3: The transaction is committed, guaranteeing the accuracy and consistency of the portfolio value.

Requirements for Reporting and Compliance:

Ensuring that all compliance and regulatory reports are founded on correct and consistent data is known as accurate and consistent reporting.
Isolation During Data Aggregation: To avoid discrepancies, transactions pertaining to data aggregation for reporting must be segregated.
For instance:
Step 1: For regulatory reporting, the system compiles trading data.
Step 2: To guarantee accuracy, make sure all data comes from committed transactions.
Step 3: The report is created and committed, guaranteeing that the data it contains is correct and consistent.

Balancing Consistency and Performance

Serializable isolation offers complete isolation to guarantee perfect data integrity, essential in stock market dealings where mistakes can cause substantial financial and reputational harm. When there is a lot of concurrency, the tight isolation requirements may cause a decrease in system throughput and an increase in latency. It can affect performance even though it guarantees the highest level of data integrity. Systems need to be built to withstand the possibility of increased resource consumption and transaction processing times. System resources may be impacted by higher locking and transaction management overhead. To lessen these consequences, effective optimization and database management strategies are needed.

The Serializable isolation level is frequently required in the stock market sector to provide the highest possible degree of data accuracy and consistency. Reliable order execution, correct portfolio management, real-time market data updates, and compliance reporting all depend on this level of isolation. Performance may be impacted, but the trade-off is justified by the necessity of perfect data integrity and dependability in a setting where mistakes might have serious financial repercussions.

Conclusion:

Transactions follow atomicity, consistency, isolation, and durability, combinely known as ACID principles to ensure data integrity and consistency in multi-user situations. Reading uncommitted modifications might result in inconsistent data, which is known as dirty reads.

Read Uncommitted isolation provides excellent efficiency, permits sloppy reads; but is perfect in situations when data correctness is not as important. Read Committed strikes a compromise between consistency and performance, eliminates dirty reads, and permits non-repeatable and phantom readings. Repeatable Read is suitable for applications requiring consistent repeated reads, it prevents unclean and non-repeatable reads while allowing phantom reads. Serializable isolation prevents all irregularities and offers the best consistency; high locking may lower performance.

Based on particular application requirements, balancing data consistency, performance, and resource utilization requires an understanding of the ability to select the proper isolation level. Choosing the appropriate level of isolation guarantees dependable and effective transaction processing, customized to meet the specific requirements of the setting.