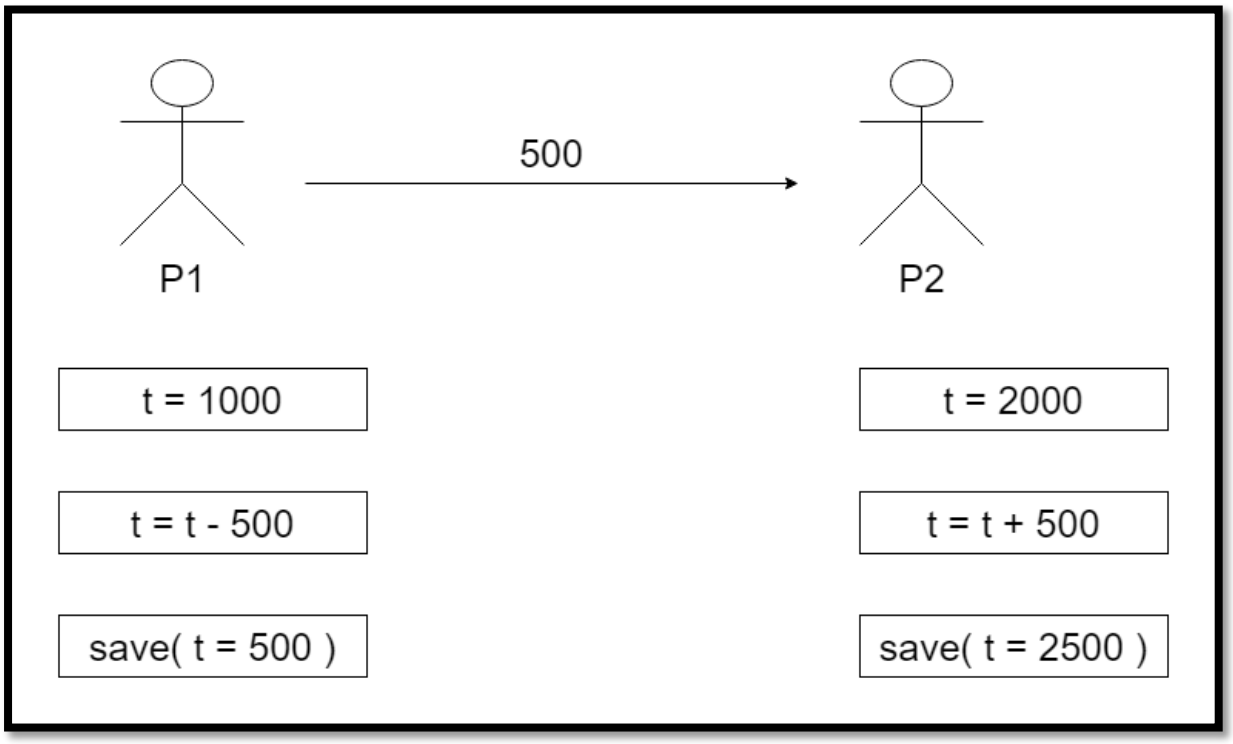
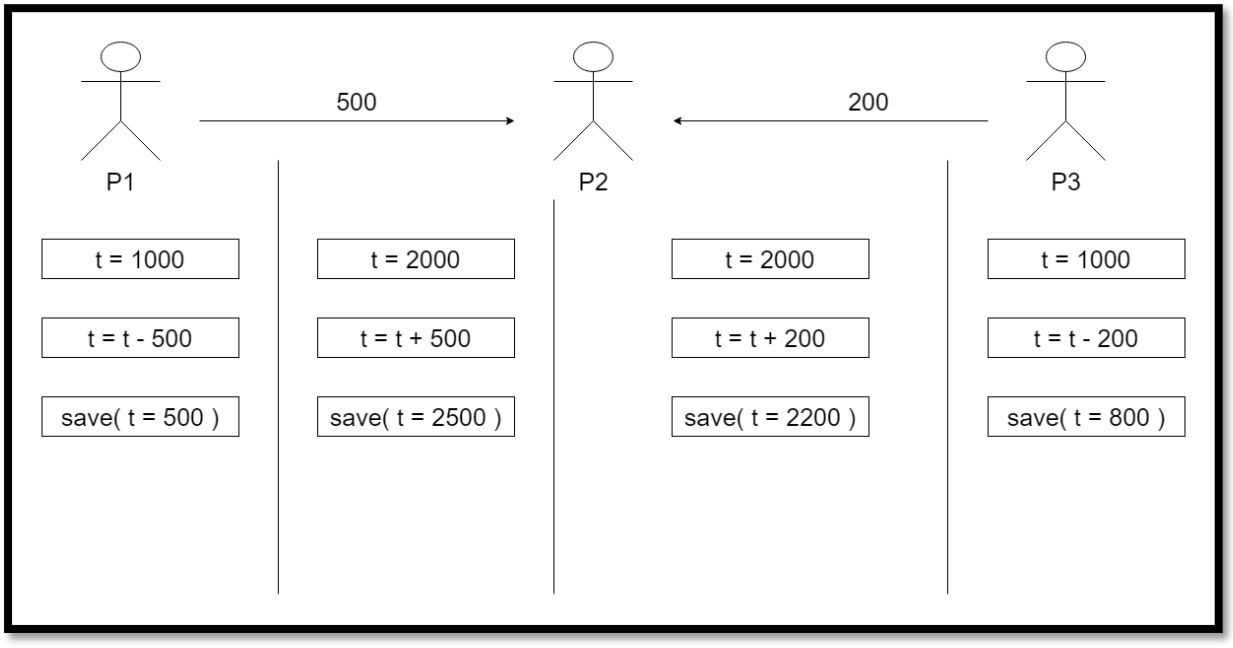
Transactions
A set of logically related operations.
In reality while performing the task, a single query is not enough.





We should solve this problem using Transaction Isolation Levels.

ACID properties are a set of four properties that ensure the reliability and consistency of database transactions, even in the presence of errors, failures, or concurrent access by multiple users. These properties are essential for maintaining data integrity in relational database systems. The acronym "ACID" stands for:

Atomicity

1. Atomicity ensures that a transaction is treated as a single, indivisible unit of work.
2. Either all the operations within a transaction are executed successfully, or none of them are.
3. If any part of the transaction fails (e.g., due to an error or exception), the entire transaction is rolled back (undone), ensuring that the database remains in a consistent state.
4. Atomicity guarantees that the database state is either left unchanged (in the case of a failure) or moved to a new consistent state (in the case of success).

Consistency

1. Consistency ensures that a transaction brings the database from one consistent state to another.
2. The database must satisfy a set of integrity constraints before and after the transaction.
3. If a transaction violates any integrity constraint, it is rolled back, and the database remains unchanged.
4. Consistency guarantees that the data remains in a valid state throughout the transaction's execution.

Isolation

1. Isolation ensures that the concurrent execution of multiple transactions does not interfere with each other.
2. Each transaction should appear as if it is executed in isolation, regardless of other concurrent transactions.
3. Isolation levels (such as Read Uncommitted, Read Committed, Repeatable Read, Serializable) define the degree to which transactions are isolated from each other, balancing data consistency and concurrency control.
4. Isolation prevents problems like dirty reads, non-repeatable reads, and phantom reads that can occur when multiple transactions access the same data simultaneously.

Durability

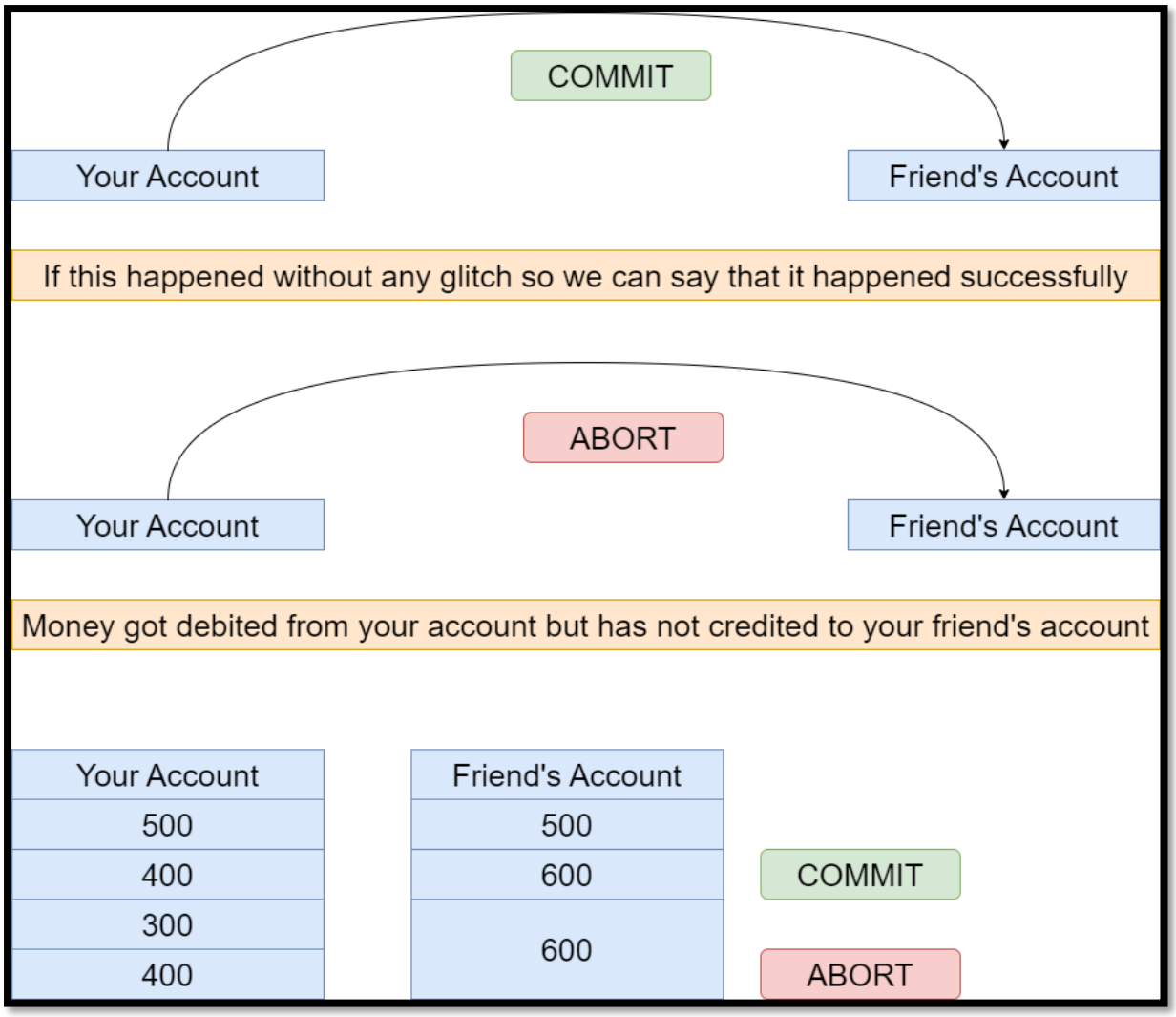
1. Durability guarantees that once a transaction is committed, its changes are permanent and will survive any subsequent failures (e.g., system crashes or power outages).
2. Committed data is stored in a durable medium (typically disk), so it can be recovered even if the system crashes.
3. Durability ensures that the database remains in a consistent state even after a system failure, and it is a critical aspect of data reliability.

Together, these ACID properties provide a framework for designing and implementing reliable and consistent database systems. While ACID transactions are essential for many use cases, they can come with performance overhead, particularly in highly concurrent systems. In some scenarios, NoSQL databases or other approaches like BASE (Basically Available, Soft state, Eventually consistent) are used to relax some ACID properties to improve system scalability and performance. The choice between ACID and BASE depends on the specific requirements and trade-offs of the application.

ACID Properties

ATOMICITY

Transaction should be performed at once or as a one whole transaction otherwise it is not performed at all.



Either the transaction has happened or not happened at all.
We know a transaction can have a set of operations in it, but we consider one whole transaction as one unit. Also it involves two operations:

1. Abort: If we abort the transaction, the changes that might have happened due to the transaction to the database aren't visible.
2. Commit: If we commit the transaction, the changes that might have happened due to the transaction to the database are visible.

CONSISTENCY

The integrity constraints of a database should be maintained before and after the database.

Now, the sum of money in the accounts before the transaction should be equal to the sum of money in the accounts after the transaction.
Therefore, we define consistency as the effect of integrity constraints on the database due to which data remains consistent before and after the transaction when it transfers the database from one state to another.
During the transaction, the database can be inconsistent.

Your Account	Friend's Account	
500	500	1000
400	600	1000
300	700	1000
1000	0	1000

Even though the account details are changed but the database as a whole is consistent. This is called integrity constraints.

ISOLATION

It ensures that parallel transactions remain consistent when they are converted into serializable form.

Multiple transactions are performed independently without any interference.

Now suppose while this transaction (mentioned above) is taking place and another transaction has started in between and it is accessing the partially updated database.

Now that transaction will encounter an inconsistent database.

Therefore, isolation ensures that the transactions are executing independently, i.e. when one transaction is being done, it won't be interrupted by the other one. Although multiple transactions can happen simultaneously, given that each transaction is unaware of the other concurrently executing transactions.

Your Account	Friend's Account	Parent's Account
500	500	500
500	1000	0
300	1100	100

DURABILITY

It ensures that data is not lost during the transactions.

Now suppose, you received a message on your mobile phone stating that Rs. 70 have been debited from your account to transfer to this account. But your friend states that he hasn't received the money yet. Now this could happen due to the software or hardware crash, but the updates to the database by the transaction has been made and they won't be lost even after the system broke down.
Therefore, this property ensures all the changes or updates to the database have been recorded and have been stored and will be never lost even if the system crashes.

```
SET AUTOCOMMIT = 0;

SET SQL_SAFE_UPDATES = 0;

SELECT * FROM PERSONS;
```

ID	FIRSTNAME	LASTNAME	CITY	ADDRESS
1	NANU	GUPTA	MANSA	PUNJAB
2	IQBAAL	SINGH	PATIALA	PUNJAB
3	TARUN	SAINI	PATIALA	PUNJAB
4	SAKSHI	SINGLA	PATIALA	PUNJAB
5	RUCHI	VISHAVKARMA	PATIALA	PUNJAB

```
START TRANSACTION;

UPDATE PERSONS SET FIRSTNAME="ANUBHAV" WHERE ID=1;

SELECT * FROM PERSONS;
```

ID	FIRSTNAME	LASTNAME	CITY	ADDRESS
1	ANUBHAV	GUPTA	MANSA	PUNJAB
2	IQBAAL	SINGH	PATIALA	PUNJAB
3	TARUN	SAINI	PATIALA	PUNJAB
4	SAKSHI	SINGLA	PATIALA	PUNJAB
5	RUCHI	VISHAVKARMA	PATIALA	PUNJAB

```
ROLLBACK;

SELECT * FROM PERSONS;
```

ID	FIRSTNAME	LASTNAME	CITY	ADDRESS
1	NANU	GUPTA	MANSA	PUNJAB
2	IQBAAL	SINGH	PATIALA	PUNJAB
3	TARUN	SAINI	PATIALA	PUNJAB
4	SAKSHI	SINGLA	PATIALA	PUNJAB
5	RUCHI	VISHAVKARMA	PATIALA	PUNJAB

```
SET AUTOCOMMIT = 0;

SET SQL_SAFE_UPDATES = 0;


SELECT * FROM PERSONS;
```

ID	FIRSTNAME	LASTNAME	CITY	ADDRESS
1	NANU	GUPTA	MANSA	PUNJAB
2	IQBAAL	SINGH	PATIALA	PUNJAB
3	TARUN	SAINI	PATIALA	PUNJAB
4	SAKSHI	SINGLA	PATIALA	PUNJAB
5	RUCHI	VISHAVKARMA	PATIALA	PUNJAB

```
START TRANSACTION;

UPDATE PERSONS SET FIRSTNAME="ANUBHAV" WHERE ID=1;

SELECT * FROM PERSONS;
```

ID	FIRSTNAME	LASTNAME	CITY	ADDRESS
1	ANUBHAV	GUPTA	MANSA	PUNJAB
2	IQBAAL	SINGH	PATIALA	PUNJAB
3	TARUN	SAINI	PATIALA	PUNJAB
4	SAKSHI	SINGLA	PATIALA	PUNJAB
5	RUCHI	VISHAVKARMA	PATIALA	PUNJAB

```
COMMIT;

ROLLBACK;


SELECT * FROM PERSONS;
```

ID	FIRSTNAME	LASTNAME	CITY	ADDRESS
1	ANUBHAV	GUPTA	MANSA	PUNJAB
2	IQBAAL	SINGH	PATIALA	PUNJAB
3	TARUN	SAINI	PATIALA	PUNJAB
4	SAKSHI	SINGLA	PATIALA	PUNJAB
5	RUCHI	VISHAVKARMA	PATIALA	PUNJAB

SQL provides transaction control commands that allow you to manage transactions within a database. These commands are essential for ensuring data integrity, consistency, and isolation in multi-user database systems. Here are the primary transaction control commands in SQL:

BEGIN TRANSACTION (or BEGIN, or START TRANSACTION)

1. Begins a new transaction explicitly. It marks the beginning of a transaction block.
2. In some database systems, the command may be simply BEGIN or START TRANSACTION.
3. Transactions are typically started automatically when you execute a SQL statement, but you can use BEGIN TRANSACTION to explicitly start a transaction if needed.
4. Example:

   ```
   BEGIN TRANSACTION;
   ```

COMMIT

1. Commits the current transaction, making all changes made during the transaction permanent.
2. Once a transaction is committed, its changes are saved to the database, and the transaction is completed.
3. Committing a transaction ensures data consistency and durability.
4. Example:

   ```
   COMMIT;
   ```

ROLLBACK

1. Rolls back the current transaction, undoing all changes made during the transaction.
2. It allows you to discard any changes made within the current transaction and return the database to its previous state.
3. Rolling back is typically used when an error occurs or when you want to cancel the effects of a transaction.
4. Example:

   ```
   ROLLBACK;
   ```

SAVEPOINT

1. Defines a savepoint within a transaction, allowing you to roll back to a specific point within the transaction.
2. You can have multiple savepoints within a transaction, and you can roll back to a specific savepoint without affecting the entire transaction.
3. Example:

   ```
   SAVEPOINT my_savepoint;
   ```

ROLLBACK TO SAVEPOINT

1. Rolls back the transaction to a specified savepoint, undoing changes made after that savepoint.
2. This allows you to maintain some parts of the transaction while undoing others.
3. Example:

   ```
   ROLLBACK TO my_savepoint;
   ```

RELEASE SAVEPOINT

1. Removes a savepoint within a transaction, freeing up resources.
2. It does not undo any changes; it simply removes the specified savepoint.
3. Example:

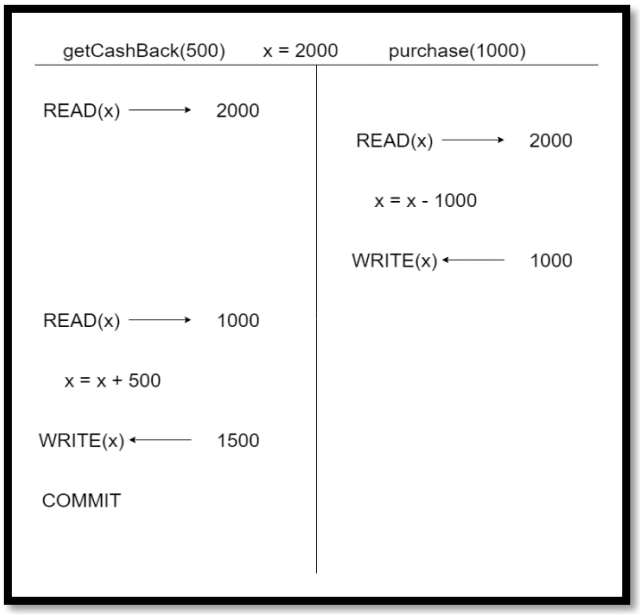
   ```
   RELEASE SAVEPOINT my_savepoint;
   ```

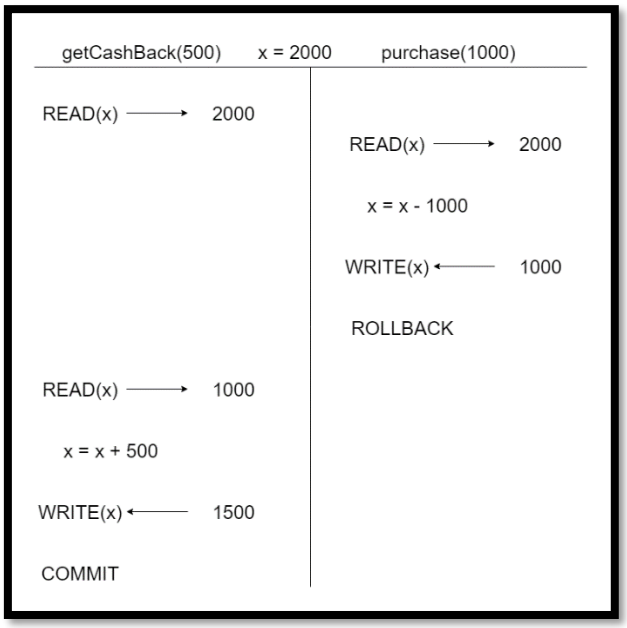
These transaction control commands are essential for ensuring that database transactions are handled correctly and reliably. They help maintain data integrity and consistency, even in the presence of errors or concurrent access by multiple users. The specific syntax and behaviour of these commands may vary slightly depending on the database management system (DBMS) you are using, so it's essential to refer to the documentation of your DBMS for precise details.

Transaction isolation levels in SQL define the degree to which one transaction is isolated from the effects of other concurrent transactions. Isolation levels help maintain data integrity and consistency in multi-user database systems. There are four standard transaction isolation levels defined by the SQL standard:

Read Uncommitted (Level 0)

1. This is the lowest isolation level.
2. Transactions at this level can read data that has been modified but not yet committed by other transactions.
3. It has the lowest level of data consistency and can lead to dirty reads, non-repeatable reads, and phantom reads.
4. Not commonly used in practice due to its potential for data integrity issues.





USER1

```
CREATE DATABASE IF NOT EXISTS SCALER;

USE SCALER;

CREATE TABLE IF NOT EXISTS INVENTORY(
      ID INT PRIMARY KEY,
      NAME VARCHAR(255),
      QTY INT
);

INSERT INTO INVENTORY VALUES(1, "WIDGET", 100);
INSERT INTO INVENTORY VALUES(2, "GADGET", 150);
INSERT INTO INVENTORY VALUES(3, "DOODAD", 75);

SHOW VARIABLES LIKE 'TRANSACTION_ISOLATION';
SELECT @@TRANSACTION_ISOLATION;
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

-- TRANSACTION 1
START TRANSACTION;

UPDATE INVENTORY SET QTY = 120 WHERE ID = 1;

SELECT * FROM INVENTORY;
```

ID	NAME	QTY
1	WIDGET	120
2	GADGET	150
3	DOODAD	75

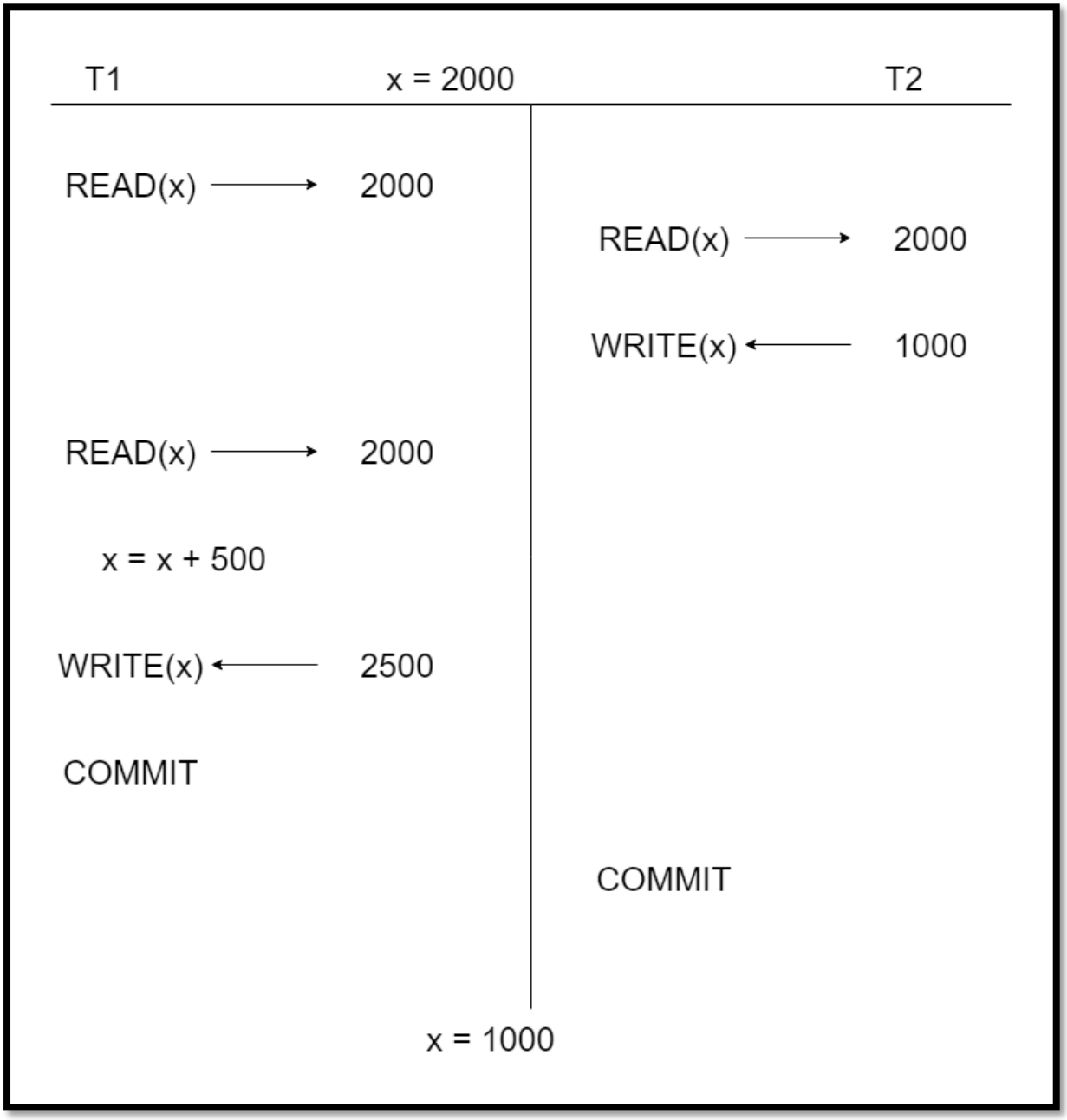
USER2

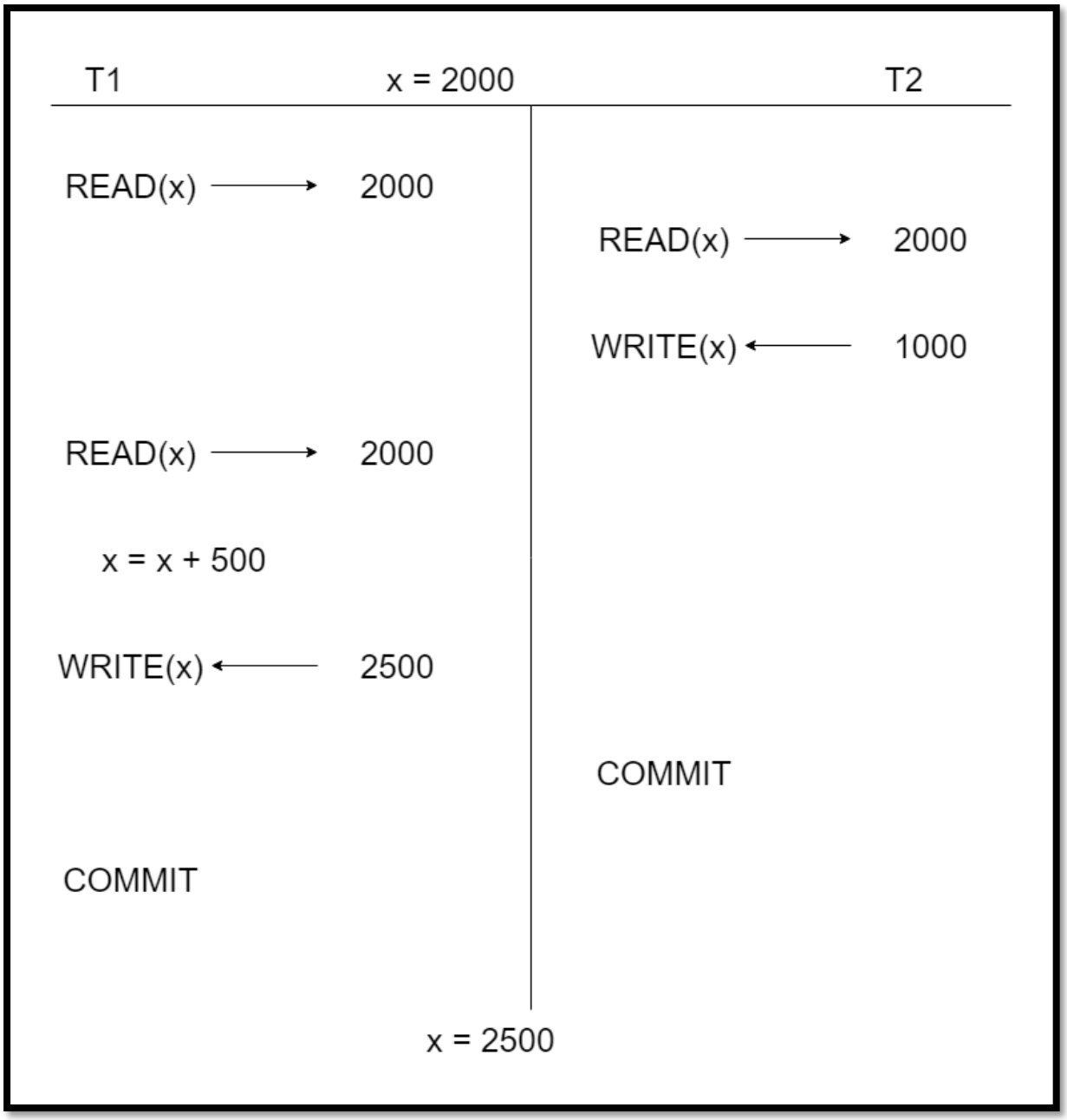
```
USE SCALER;

SHOW VARIABLES LIKE 'TRANSACTION_ISOLATION';
SELECT @@TRANSACTION_ISOLATION;
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT * FROM INVENTORY;
```

ID	NAME	QTY
1	WIDGET	120
2	GADGET	150
3	DOODAD	75

Read Committed (Level 1)

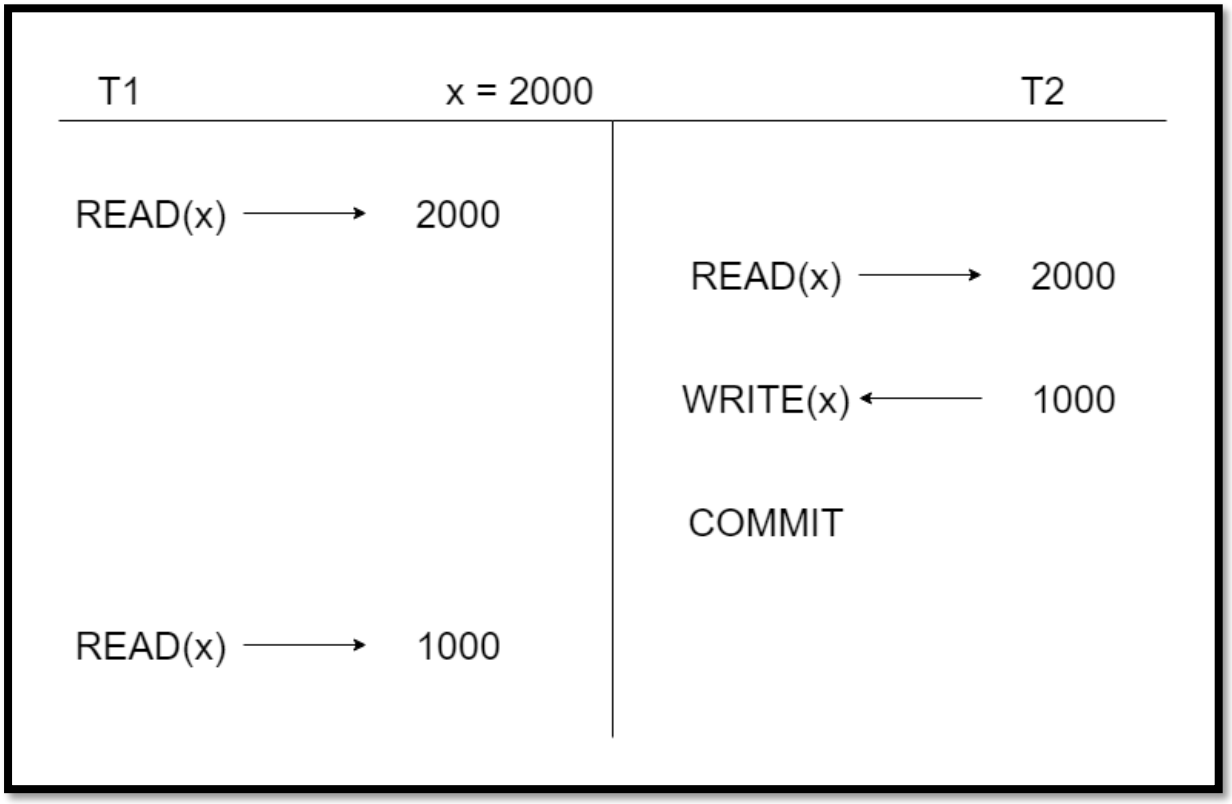
1. This is the default isolation level in most database systems.
2. Transactions at this level can only read committed data.
3. It prevents dirty reads but may still allow non-repeatable reads and phantom reads.
4. It provides a higher level of data consistency compared to Read Uncommitted.




T1
x = 2000
T2
READ(x) → 2000
READ(x) → 2000
WRITE(x) ← 1000
READ(x) → 2000
x = x + 500
WRITE(x) ← 2500
COMMIT
COMMIT
x = 2500

NON-REPEATABLE READ

A "non-repeatable read" is a phenomenon that can occur in a database when a transaction reads a particular piece of data, and during the course of the transaction, another concurrent transaction modifies or deletes that data. This results in the first transaction seeing different values for the same data when it is read again within the same transaction. Non-repeatable reads are typically prevented or minimized by using appropriate transaction isolation levels.



USER1

```
CREATE DATABASE IF NOT EXISTS SCALER;

USE SCALER;

CREATE TABLE IF NOT EXISTS INVENTORY(
      ID INT PRIMARY KEY,
      NAME VARCHAR(255),
      QTY INT
);

INSERT INTO INVENTORY VALUES(1, "WIDGET", 100);
INSERT INTO INVENTORY VALUES(2, "GADGET", 150);
INSERT INTO INVENTORY VALUES(3, "DOODAD", 75);

SHOW VARIABLES LIKE 'TRANSACTION_ISOLATION';
SELECT @@TRANSACTION_ISOLATION;
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- TRANSACTION 1
START TRANSACTION;

UPDATE INVENTORY SET QTY = 120 WHERE ID = 1;

SELECT * FROM INVENTORY;
```

ID	NAME	QTY
1	WIDGET	120
2	GADGET	150
3	DOODAD	75

```
COMMIT;
```

USER2

```
USE SCALER;

SHOW VARIABLES LIKE 'TRANSACTION_ISOLATION';
SELECT @@TRANSACTION_ISOLATION;
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT * FROM INVENTORY;
```

ID	NAME	QTY
1	WIDGET	120
2	GADGET	150
3	DOODAD	75

USER1

```
CREATE DATABASE IF NOT EXISTS SCALER;

USE SCALER;

CREATE TABLE IF NOT EXISTS INVENTORY(
      ID INT PRIMARY KEY,
      NAME VARCHAR(255),
      QTY INT
);

INSERT INTO INVENTORY VALUES(1, "WIDGET", 100);
INSERT INTO INVENTORY VALUES(2, "GADGET", 150);
INSERT INTO INVENTORY VALUES(3, "DOODAD", 75);

SHOW VARIABLES LIKE 'TRANSACTION_ISOLATION';
SELECT @@TRANSACTION_ISOLATION;
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- TRANSACTION 1
START TRANSACTION;

UPDATE INVENTORY SET QTY = 120 WHERE ID = 1;

SELECT * FROM INVENTORY;
```

ID	NAME	QTY
1	WIDGET	120
2	GADGET	150
3	DOODAD	75

USER2

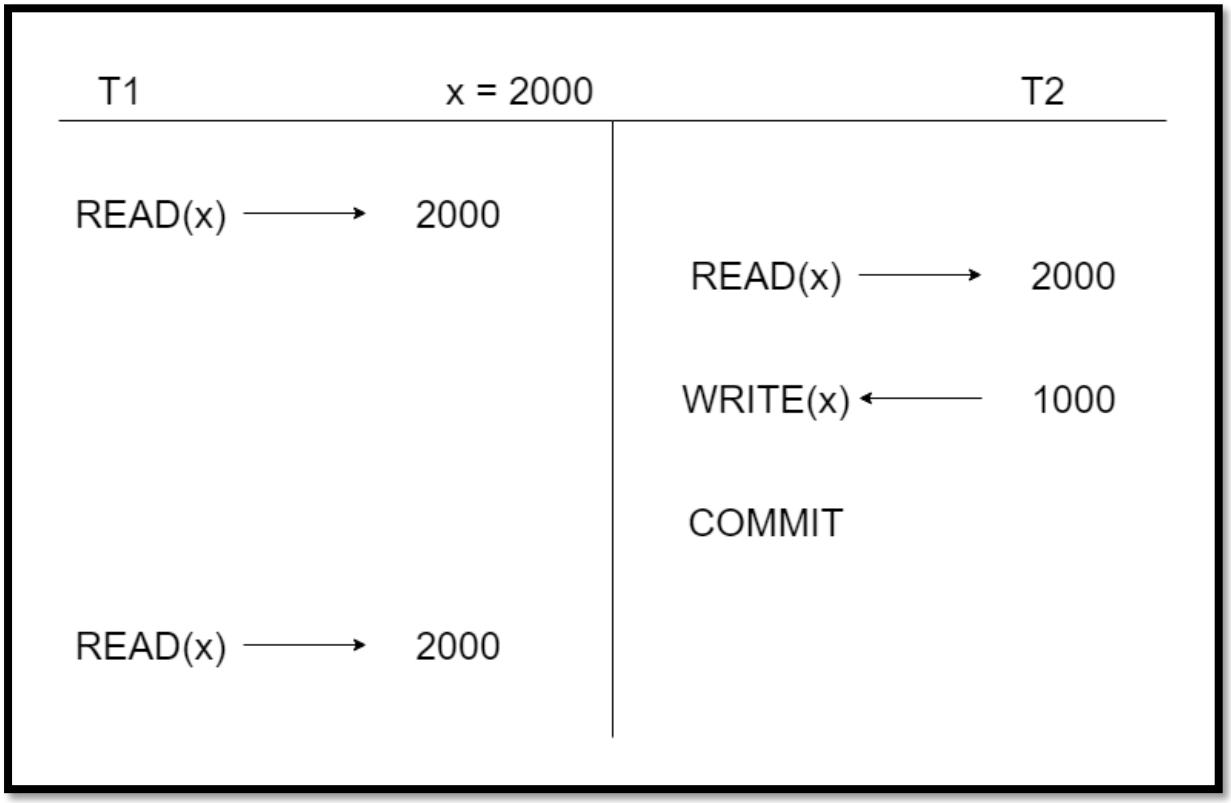
```
USE SCALER;

SHOW VARIABLES LIKE 'TRANSACTION_ISOLATION';
SELECT @@TRANSACTION_ISOLATION;
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT * FROM INVENTORY;
```

ID	NAME	QTY
1	WIDGET	100
2	GADGET	150
3	DOODAD	75

READ COMMIT solves dirty read problem but introduces the problem of lost update or dirty write.

Repeatable Read (Level 2)

1. It is the default read in many DBMS especially in MYSQL.
2. In this isolation level, a transaction sees a consistent snapshot of the data as of the start of the transaction.
3. It prevents dirty reads and non-repeatable reads.
4. Lost update or dirty write can still occur.
5. It provides a higher level of isolation than Read Committed.



Serializable (Level 3)

1. This is the highest isolation level.
2. It ensures that only one transaction run at a time which means no concurrency.
3. It provides full isolation from other transactions, ensuring that no other transactions can read or modify data that the current transaction is working with.
4. It prevents dirty reads, non-repeatable reads, and phantom reads.
5. Achieving this level of isolation often involves locking, which can lead to performance issues in high-concurrency environments.

Keep in mind that the actual behaviour and implementation of these isolation levels may vary between different database management systems (DBMS). Some databases offer additional isolation levels or variations of the standard levels, and some may implement isolation differently under the hood.

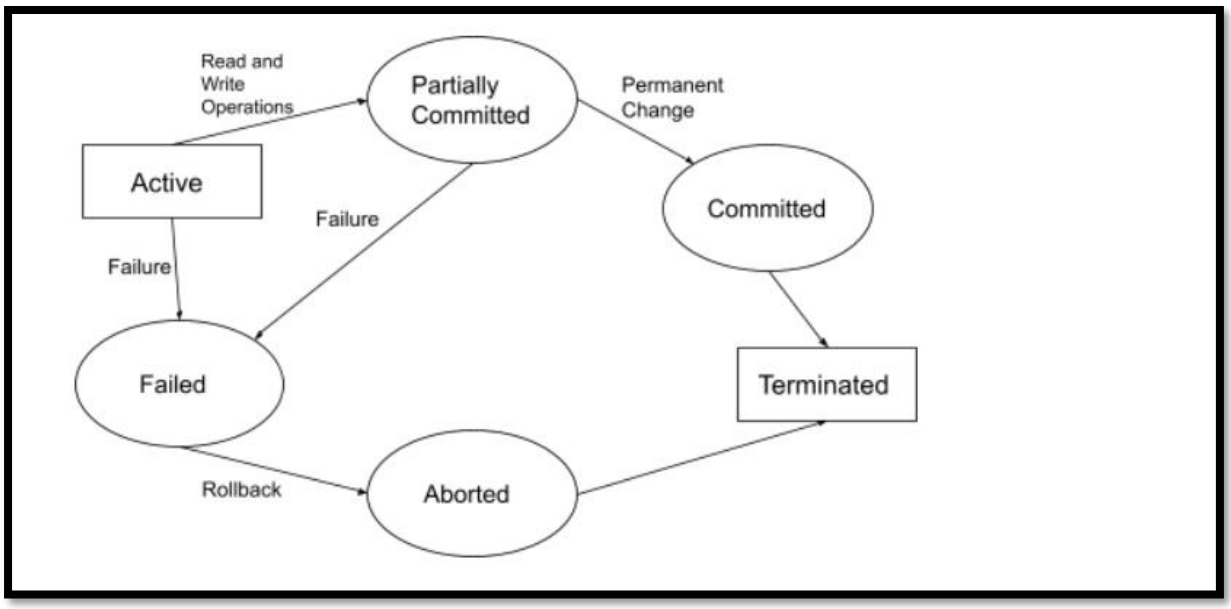
You can set the isolation level for a transaction using SQL statements specific to your DBMS. For example, in PostgreSQL, you can use the SET TRANSACTION ISOLATION LEVEL statement, and in MySQL, you can use the SET TRANSACTION statement to specify the desired isolation level for a transaction.

Choosing the appropriate isolation level depends on your application's requirements, balancing the need for data consistency and concurrency. It's essential to understand the implications and trade-offs associated with each isolation level to make an informed decision.

TRANSACTION STATES

A transaction goes through numerous states throughout its life cycle. These states are known as transaction states. The states are:

1. Active state
2. Partially committed state
3. Committed state
4. Failed state
5. Aborted state
6. Terminated state



Active State

The very first state of the life cycle of transaction, all the read and write operations are being performed and if they execute without any error the transaction comes to ‘partially committed’ state, although if any error then it leads to ‘failed’ state. Note: All the changes made by the transaction now are stored in the buffer in main memory.

Partially Committed State

After the last command of the transaction is executed the changes are saved in the buffer in Main Memory. If the changes made are permanent on the Database then the state will transfer to the ‘committed’ state and if there's any kind of failure it will go to the ‘failed’ state.

Committed Sate

When the updates are made permanent on the database. Then the transaction is said to be in Committed state. Note: Rollback can’t be done from here. At this state, a new consistent state is achieved by the database.

Failed State

When a transaction is being executed and some failure occurs due to which it becomes impossible to continue the execution. Transaction enters into a failed state. We can come to this state from an Active or Partially committed state.

Aborted State

From Failed State, when all the changes made in the buffer are reversed, and now that transaction needs to Rollback completely it enters Aborted state.

Terminated State

This is the final state of the life cycle of the transaction. It is where it ends. A transaction can come to this state only from a ‘committed’ or ‘aborted’ state.

After completing the cycle, now the database is consistent and is ready for new transactions.