TRANSACTION MANAGEMENT AND CONCURRENCY CONTROL
UNIT 2

What is a transaction?

A simple understanding:

A transaction is a completed agreement between a buyer and a seller to exchange goods or services

Examples of transactions are :

- Paying a supplier for services rendered or goods delivered.
- Paying an employee for hours worked
- A bank transaction is a record of money that has moved in and out of your bank account.

In Databases : What is a TRANSACTION?

A Transaction in a database is any operation that reads from or writes data to a database.

The various statements that are used to perform read or write operation on a database or to access a database are :

SELECT : to read data from a database

UPDATE : to change the value of fields (attributes) of a table

INSERT : to add rows to a table

A transaction can be:

- a single or a series of SELECT statement
- a single or a series of UPDATE statement
- a single or a series of INSERT statement
- A combination of SELECT, UPDATE or INSERT statements.

So a Database Transaction is a logical unit of processing in a DBMS which involves one or more database access operation (SQL statement).

We can say a transaction is a set of **logically related** operations. For example, you are transferring money from your bank account to your friend’s account, the set of operations would be like this:

1. Read your account balance	READ Y
2. Deduct the amount from your balance	WRITE Y = Y -500
3. Write the remaining balance to your account	WRITE Y
4. Read your friend’s account balance	READ F
5. Add the amount to his account balance	WRITE F = F+500
6. Write the new updated balance to his account	WRITE F

One logical unit of work or One transaction

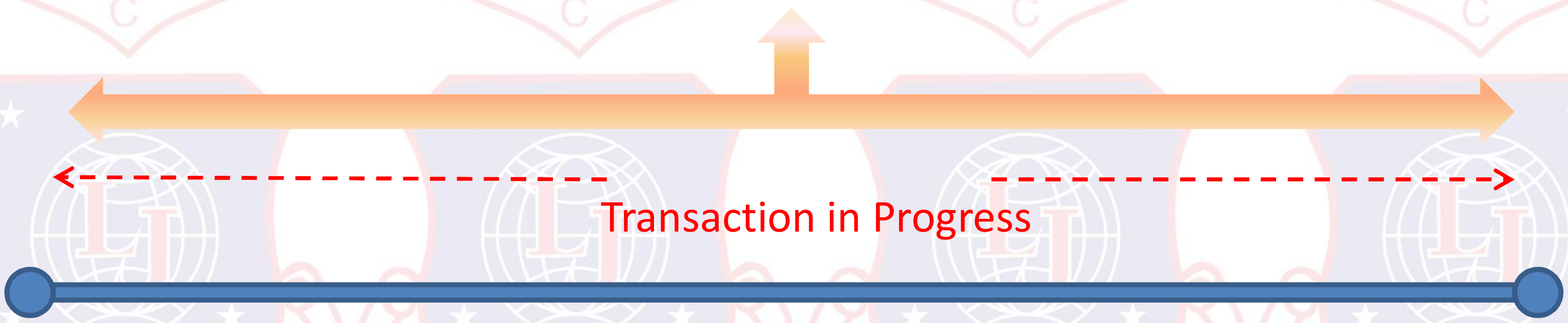
Here the set of above statements are executed to execute a transaction of transferring Rs 500 to your friends account.

THIS IS ONE TRANSACTION.
THIS IS ONE LOGICAL UNIT OF WORK

A transaction in a database must be either ENTIRELY COMPLETED or ENTIRELY ABORTED

There can not be any intermediate state of a transaction.

INCONSISTENT STATE



Beginning of Transaction

```
READ   Y
WRITE Y = Y -500
WRITE  Y
READ   F
WRITE  F = F+500
WRITE  F
```

End of Transaction

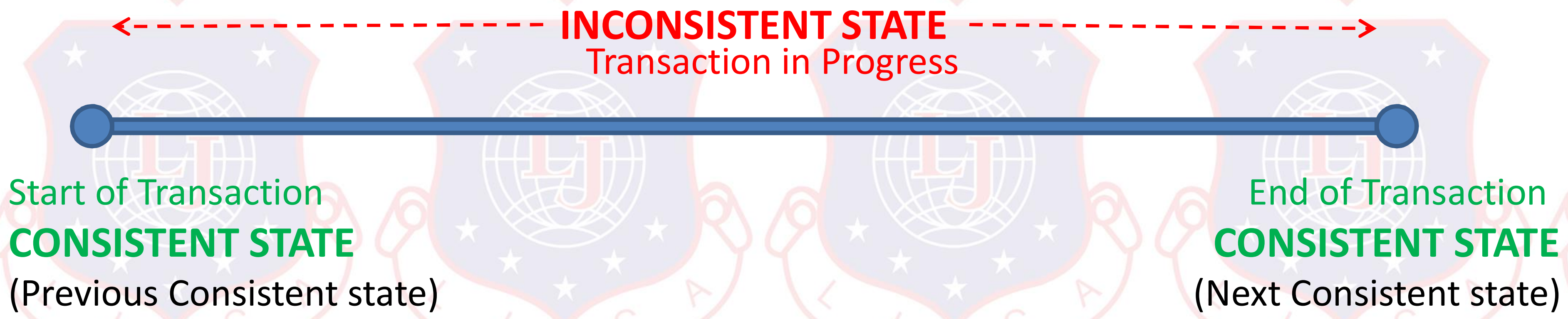
(Completed Transaction)

CONSISTENT STATE

CONSISTENT STATE

All these SQL statements must be executed successfully. If any statement fails the entire transaction is ROLLED BACK to the previous consistent state (state before the transaction started)

A successful transaction takes a database from one consistent state to another consistent state. Or we can say from a previous consistent state to the next consistent state.



If this happens all data integrity rules are satisfied.

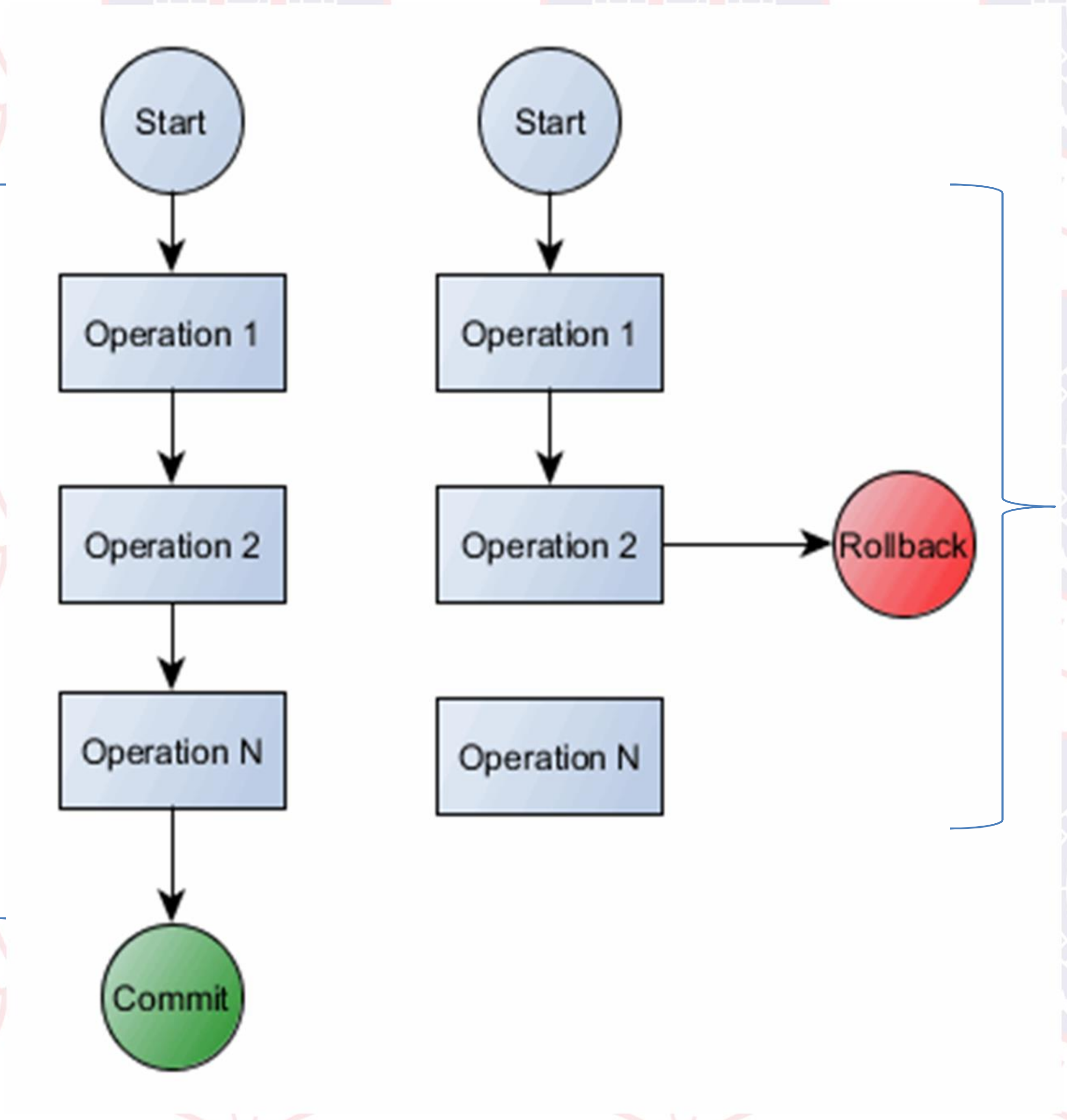
Successful Transaction

~~UNSuccessful~~ Transaction

Consistent State

InConsistent State

Consistent State



Consistent State

InConsistent State

~~Consistent State~~

All Transactions DO NOT Update the database.

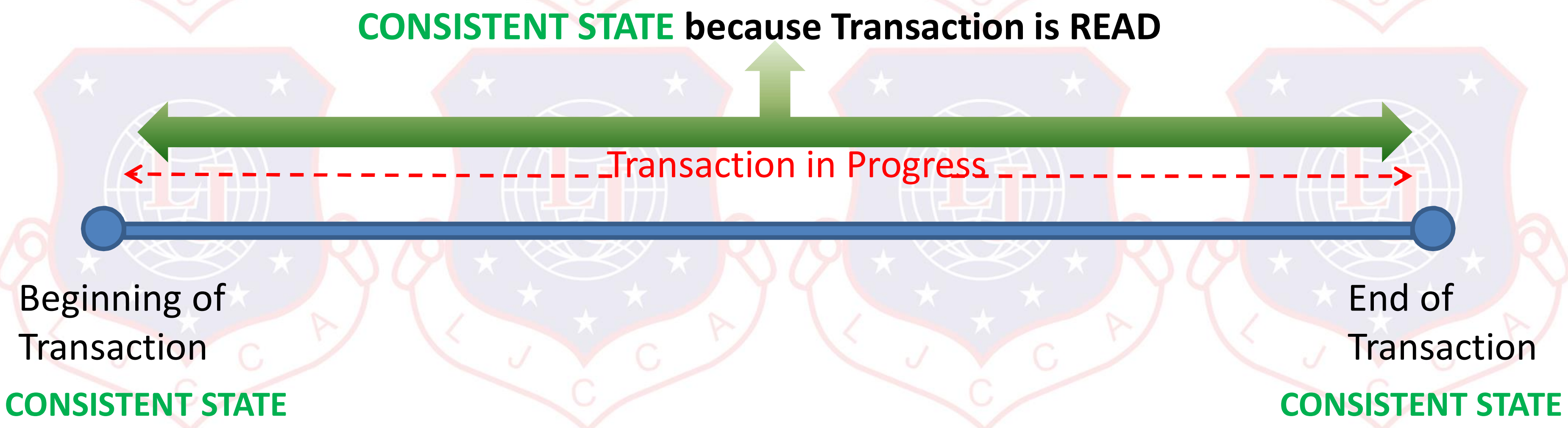
For eg: In an employee table (Employee) if we want to find the salary of an employee whose employee number is 1057 then the query is :

SELECT Empno, Salary FROM Emplyee WHERE Empno = 1057;

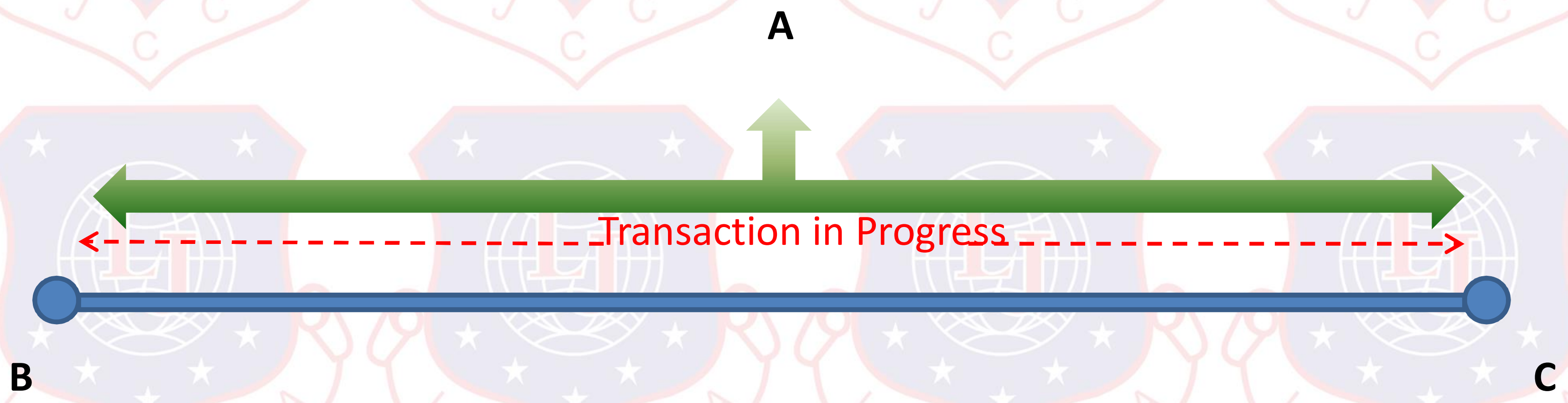
This is a transaction but NO changes are made to the database .

This transaction only accesses the database.

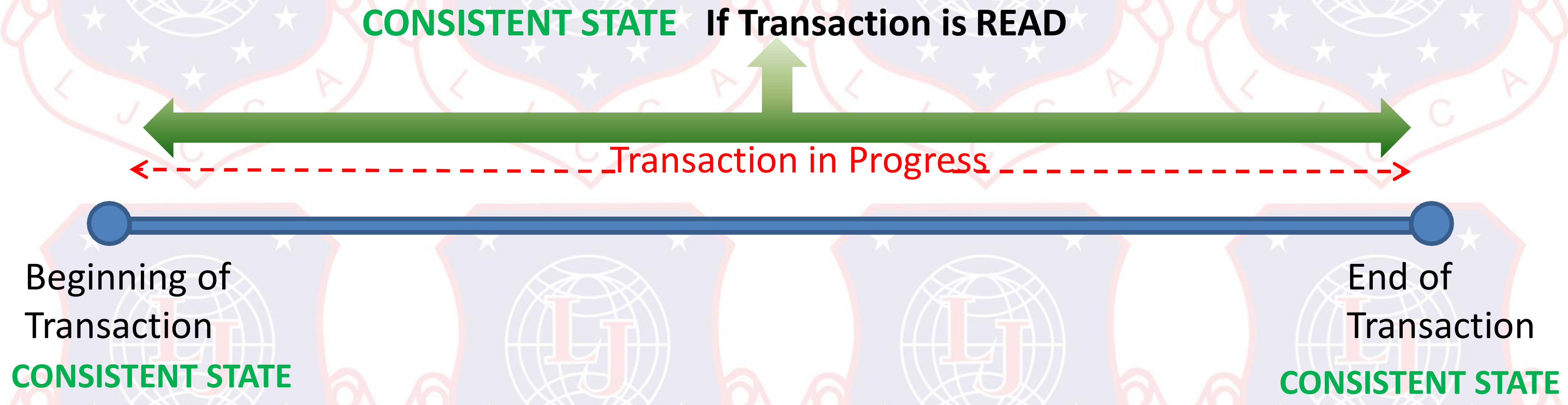
In such type of a transaction the database was in a consistent state before the database was accessed for READ transaction, and it remained in a consistent state as there were no changes made to the database.

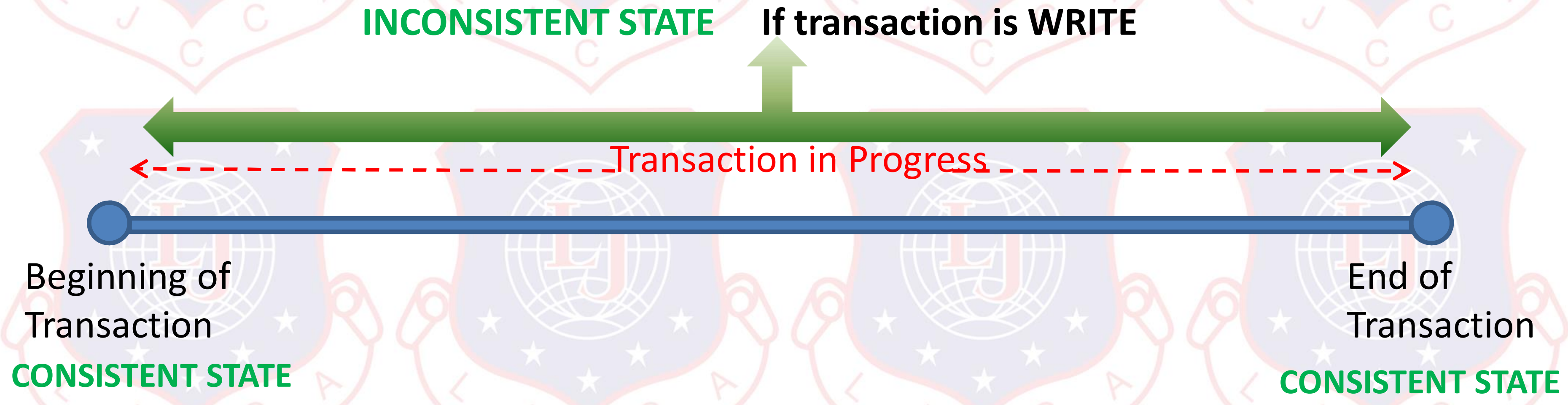


READ Transaction



If this is a READ transaction what is A , B and C


CONSISTENT STATE
If Transaction is READ
Transaction in Progress
Beginning of Transaction
CONSISTENT STATE
End of Transaction
CONSISTENT STATE


INCONSISTENT STATE
If transaction is WRITE
Transaction in Progress
Beginning of Transaction
CONSISTENT STATE
End of Transaction
CONSISTENT STATE

A transaction can be a single SQL statement or a group of related SQL statements.

For eg:
Consider the group of SQL statements:

Before execution starts : PREVIOUS CONSISTENT STATE

During exection
INCONSISTENT STATE

```
INSERT INTO t1 VALUES (.....................)

INSERT INTO t2 VALUES (....................)

UPDATE t3 SET ............

UPDATE t4 SET ..............

INSERT INTO t5 VALUES (......................)

COMMIT;
```

One transaction

Execution ends : NEXT CONSISTENT STATE

A transaction can be a single SQL statement or a group of related SQL statements.

For eg:
Consider the group of SQL statements:

Before execution starts : PREVIOUS CONSISTENT STATE

INSERT INTO t1 VALUES (......................)

INSERT INTO t2 VALUES (.....................)

During exection
INCONSISTENT STATE

UPDATE t3 SET ....

UPDATE t4 SET ....

POWER FAILURE
R O L L B A C K

PROPERTIES OF TRANSACTION

ACIDS

A = Atomicity	A transaction is either successfully completed or aborted
C =Consistency	A transaction takes a database from one consistent state to another consistent state
I = Isolation	Data being used by one transaction cannot be used by another transaction at the same time.
D = Durability	Once a transaction changes are committed they cannot be undone even in case of system failure.
S = Serializability	It ensures that concurrent execution of transactions gives consistent results.

In a single User database system the properties of ISOLATION and SERIALIZABILITY are automatically achieved as only one transaction at a time is executed.

In a multiuser database **many concurrent transactions** (multiple transactions at the same time over the same data set) are carried out. In this case the DBMS should take care of ISOLATION and SERIALIZABILITY along with other properties so that database maintains consistency and integrity. This is done by **Concurrency Control.**

Transaction Management with SQL

Every transaction which starts must continue until one of the following happens:

1. A COMMIT statement is reached. A Commit ends the transaction and permanently saves all the changes in the database. **It takes the database to the next consistent state.**

2. A ROLLBACK statement is reached. This aborts (undo) all changes and the **database comes back to the previous consistent state.**

3. The end of program is encountered. This is same as COMMIT. All changes are permanently saved and the **database goes to the next consistent state.**

4. The program is terminated. All changes are aborted and the **database is rolled back to the previous consistent state.**

TRANSACTION LOG

DBMS uses a transaction log to keep track of all transactions.

The information in the transaction log is used for recovery in case of

a ROLLBACK
program abnormal termination
system failure (network or disk crash)

This transaction log is used whenever the database has to be ROLLBACK to the previous consistent state.

In case of a system failure two things happen:

1. All **uncommitted** transactions are **rolled back**
2. All **committed** transactions which were not written to the physical database are **rolled forward** (physically written on the database)

The transaction log keeps track of the following things:

- Record of the beginning of transaction
- For each transaction it keeps track of :
  - Type of operation (UPDATE, DELETE or INSERT)
  - Name of the Table affected by the transaction
  - The “before” and “ after” values of any field which is updated
  - Pointers to the next and previous transaction log entries
- The end of the transaction (COMMIT)

Example of a Transaction Log

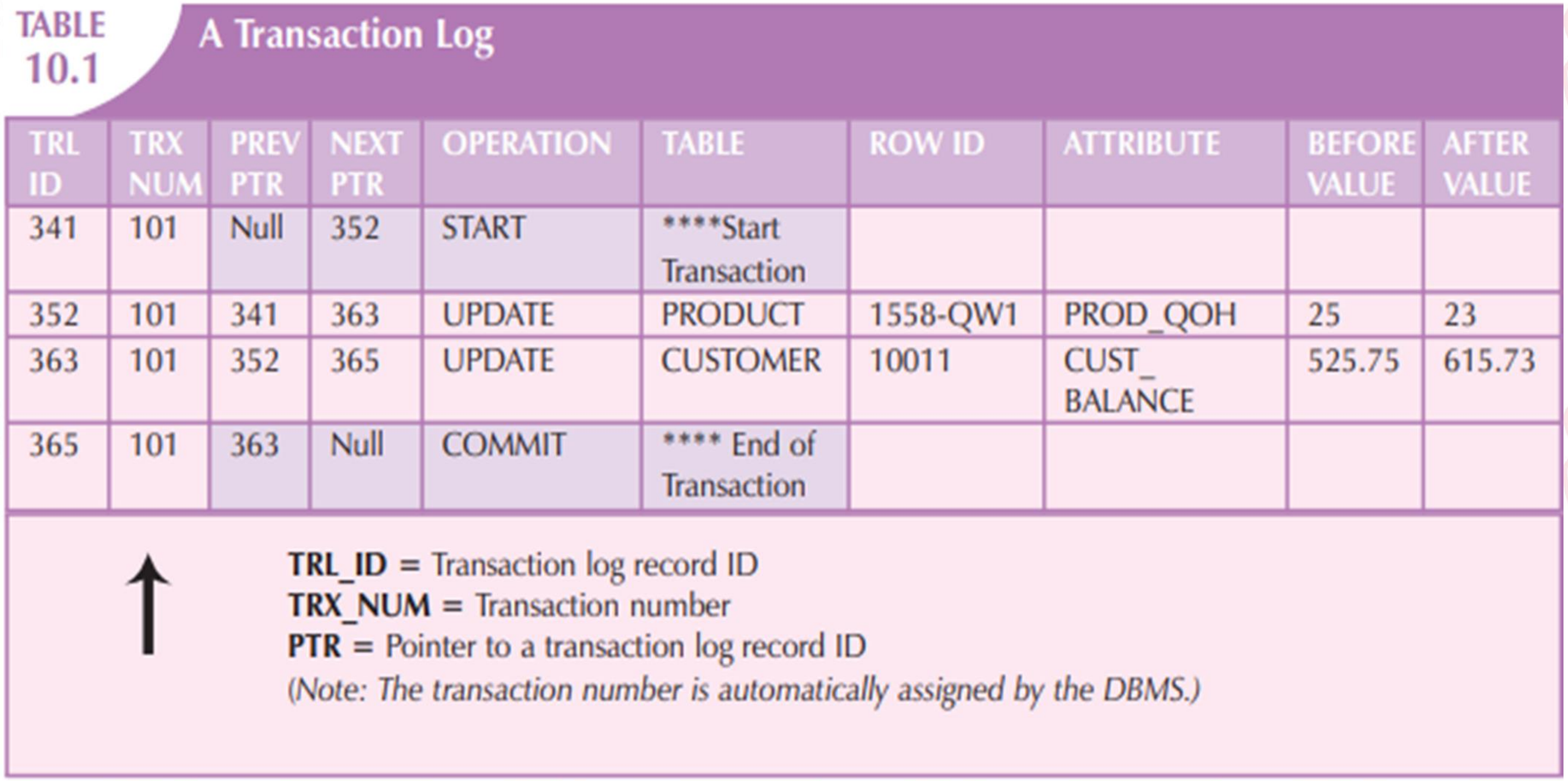


TABLE 10.1 A Transaction Log

TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	PRODUCT	1558-QW1	PROD_QOH	25	23
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_ BALANCE	525.75	615.73
365	101	363	Null	COMMIT	**** End of Transaction				

↑

TRL_ID = Transaction log record ID
TRX_NUM = Transaction number
PTR = Pointer to a transaction log record ID
(Note: The transaction number is automatically assigned by the DBMS.)

Concurrency Control

What is Concurrency Control?

In a multiuser environment there can be multiple transactions getting executed simultaneously over a shared database. The coordination of these simultaneous execution of transactions in a multiuser database system is known as Concurrency Control.

The need for concurrency control is to ensure **SERIALIZABILITY** in a multiuser environment

If concurrency is not controlled , it can lead to several problems like:

1. **Lost Updates**
2. **Uncommitted Data**
3. **Inconsistent Retrieval**

Problem of LOST UPDATES if concurrency is not controlled

This is a situation where two concurrent transactions are updating the same data, and one of the updates is **lost** as it is overwritten by the other transaction.

Example :

Suppose if we a PRODUCT table, with a field : Product Quantity on hand (**P_QOH**)

If the **P_QOH** of a product is **35,** and there are two transactions T1 and T2 which update the P_QOH like :

Transaction **T1**: Buy 100 units	**P_QOH = P_QOH + 100**
Transaction **T2 :** Sell 30 units	**P_QOH = P_QOH - 30**

There are now two possibilities how these transactions are executed:

1. Serial Execution of Transaction

Time	Transaction	Action	Value
1	T1	Read P_QOH	35
2	T1	P_QOH= P_QOH + 100	
3	T1	Write P_QOH	135
4	T2	Read P_QOH	135
5	T2	P_QOH = P_QOH-30	
6	T2	Write P_QOH	105

Final value of P_QOH is 105

1. Serial Execution of Transaction

Time	Transaction	Action	Value
1	T1	Read P_QOH	35
2	T1	P_QOH= 35 + 100	
3	T1	Write P_QOH	135
4	T2	Read P_QOH	135
5	T2	P_QOH = 135-30	
6	T2	Write P_QOH	105

Final value of P_QOH is 105

2. LOST Updates

Time	Transaction	Action	Value
1	T1	Read P_QOH	35
2	T2	Read P_QOH	35
3	T1	P_QOH = 35 + 100	
4	T2	P_QOH = 35-30	
5	T1	Write P_QOH	135
6	T2	Write P_QOH	5

Final value of P_QOH is 5

Problem of UNCOMMITTED DATA if concurrency is not controlled

This is a situation where the Isolation property is violated.
Here T1 and T2 are two transactions which are executed concurrently.

Transaction **T1**: Buy 100 units	**P_QOH = P_QOH + 100**
Transaction **T2 :** Sell 30 units	**P_QOH = P_QOH - 30**

Here T1 transaction executed and then rolled back, T2 transaction executes normally

The problem of uncommitted data arises when T1 rolls back after T2 accesses the data.

In case of a correct execution : the value of P_QOH is changed as:

Correct Execution of Transaction

Time	Transaction	Action	Value
1	T1	Read P_QOH	35
2	T1	P_QOH= 35 + 100	
3	T1	Write P_QOH	135
4	T1	****ROLLBACK**	
5	T2	Read P_QOH	35
6	T2	P_QOH = 35-30	
7	T2	Write P_QOH	5

Correct final value of P_QOH is 5

UNCOMMITTED DATA PROBLEM

Time	Transaction	Action	Value
1	T1	Read P_QOH	35
2	T1	P_QOH= 35 + 100	
3	T1	Write P_QOH	135
4	T2	Read P_QOH	135
5	T2	P_QOH = 135-30	
6	T1	****ROLLBACK**	
6	T2	Write P_QOH	105

Here transaction T2 is reading uncommitted data

Final value of P_QOH is 105

Problem of INCONSISTENT RETRIEVAL if concurrency is not controlled

This problem arises when one transaction access data before and after other transaction finish working with this data.

For example if there are two transactions: T1 and T2.
T1 is calculating the sum of a set of data and at the same time Transaction T2 is changing the same data.
In this case a transaction might read some data before it is changed and some data after it is changed. This gives inconsistent results.

Example :
T1 transaction calculates the total of QOH for all the products in PRODUCT table
T2 transaction updates the QOH of two products

Transaction T1	**Transaction T2**
Select sum(QOH) from PRODUCT	Update PRODUCT set QOH = QOH + 10 where prod_code = P1 Update PRODUCT set QOH = QOH -10 where prod_code = P2 COMMITT;

Transaction T1 is as :

Prod code	QOH (Before)	QOH (After)
P1	8	8
P2	32	40
P3	15	55
P4	23	78
P5	8	86
P6	6	92

Total QOH = 92

Under normal circumstances if transaction T1 occurs first Total QOH = 92

Transaction T2 is as :

Prod code	QOH (before)	QOH (after)
P1	8	8
P2	32	32
P3	15	15+ 10 = 25
P4	23	23 -10 = 13
P5	8	8
P6	6	6

Total QOH = 92

Time	Transaction	Action	Value of QOH	Total
1	T1	Read QOH for Prod_code = P1	8	8
2	T1	Read QOH for Prod_code = P2	32	40
3	T2(update)	Read QOH for Prod_code = P3	15	
4	T2	QOH= QOH+10 (15+10)		
5	T2	Write QOH for prod_code = P3	25	
6	T1	Read QOH for Prod_code = P3	25	65
7	T1	Read QOH for Prod_code = P4	23	88
8	T2	Read QOH for Prod_code = P4	23	
9	T2	QOH = QOH -10 (23-10)		
10	T2	Write QOH for Prod_code=P4	13	
11	T2	**Commit**		
12	T1	Read QOH for prod_code=P5	8	96
13	T1	Read QOH for Prod_code=P6	6	102

Here the total **QOH is 102 which is wrong**

The Scheduler

If two transactions T1 & T2 access unrelated data, there is no problem and the order of execution of the two transactions is irrelevant.

But if two transactions T1 & T2 access same data, problems can arise if there is no serializability. Also the order of execution of the two transaction is important as:

T1 then T2 or T2 then T1 both these orders can give different results.

How is the order of execution determined?

Who determines the order?

This is done by the part of the DBMS known as the SCHEDULER

Scheduler is the part of the DBMS which determines the order in which the operations within concurrent transactions will be executed.
It interleaves the execution of operations to ensure serializability and isolation properties of a transaction.

First the DBMS finds out which transactions are serializable.

Serializable transactions are those transactions which if interleaved or executed one after the other will produce the same results

Like:

Interleaved	One after the other
T1	T1
T2	T1
T3	T1
T1	T2
T2	T2
T3	T3
T1	T3

The Scheduler also takes care of the efficient use of the CPU.
It takes care that the CPU time is not wasted.

Scheduler takes care of isolation by not allowing two transactions to update the same data value at the same time.
So it has to take care of conflicting operations like :

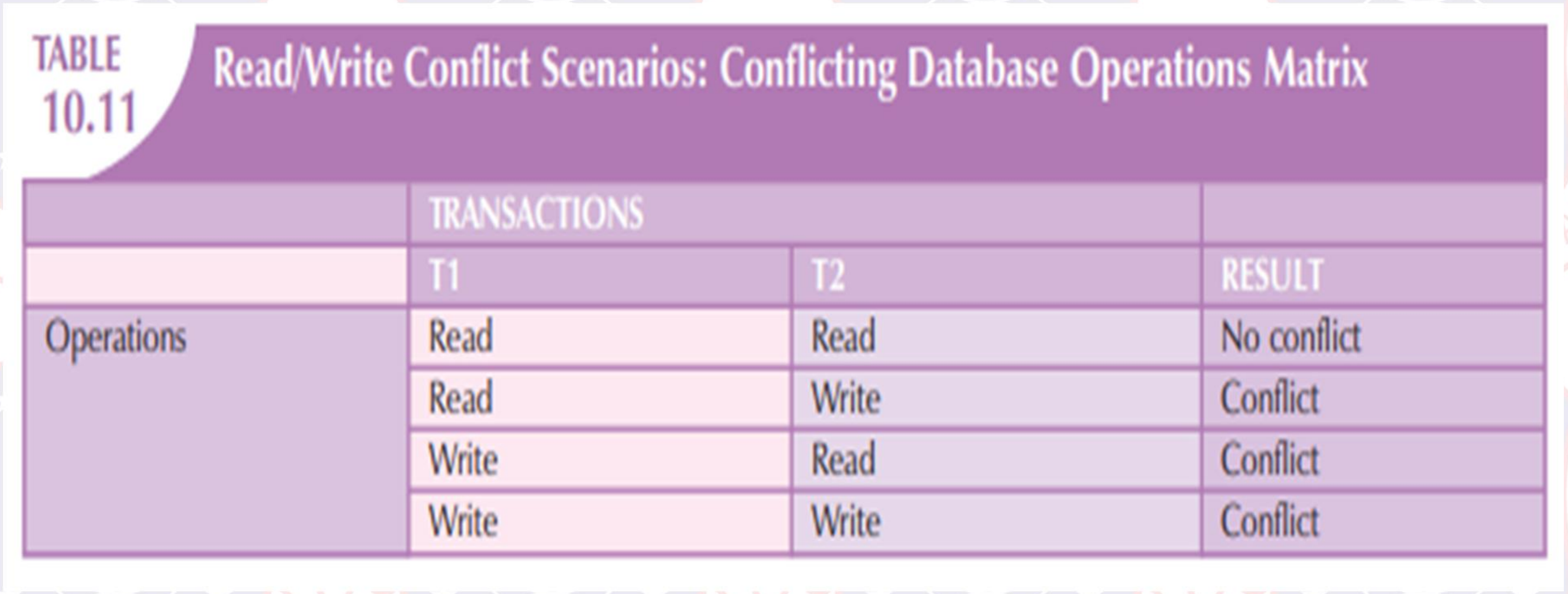


TABLE 10.11 Read/Write Conflict Scenarios: Conflicting Database Operations Matrix

	TRANSACTIONS		
	T1	T2	RESULT
Operations	Read	Read	No conflict
	Read	Write	Conflict
	Write	Read	Conflict
	Write	Write	Conflict

Conflicting operations in concurrent transactions are scheduled using the following methods :

1. Locking _ Method
2. Time_ stamping Method
3. Optimistic _ Method

1. LOCKING METHOD

In case of Concurrent transaction LOCKS are used to ensure that the data item is used by only one transaction at a time.

For example if transaction T1 is using a data item , Transaction T2 will be denied access to that data item.

If T1 wants to access a data item, it first has to go through the following steps:

```
check if the data item to be free from any other transaction using it
if free
    lock the data item
    use it (execute the transaction)
    release the lock (unlock) so that some other transaction can use it
else
    wait
```

All the lock information and entire managing of locks is done by a **LOCK MANAGER**

LOCK GRANULARITY

Lock granularity means the level of locks

Like :

- Database level locks
- Table level locks
- Page level locks
- Row level lock
- Field level lock

DATABASE LEVEL LOCKS

Here the entire database is locked by a transaction.
Imagine a **Payroll** database having two tables : **Table A and Table B**
Transaction T1 uses Table A
In case of database level locks transaction T1 locks the entire database (Table A & B)

Disadvantage: very slow access in case of multiple transactions

FIGURE 10.3 Database-level locking sequence

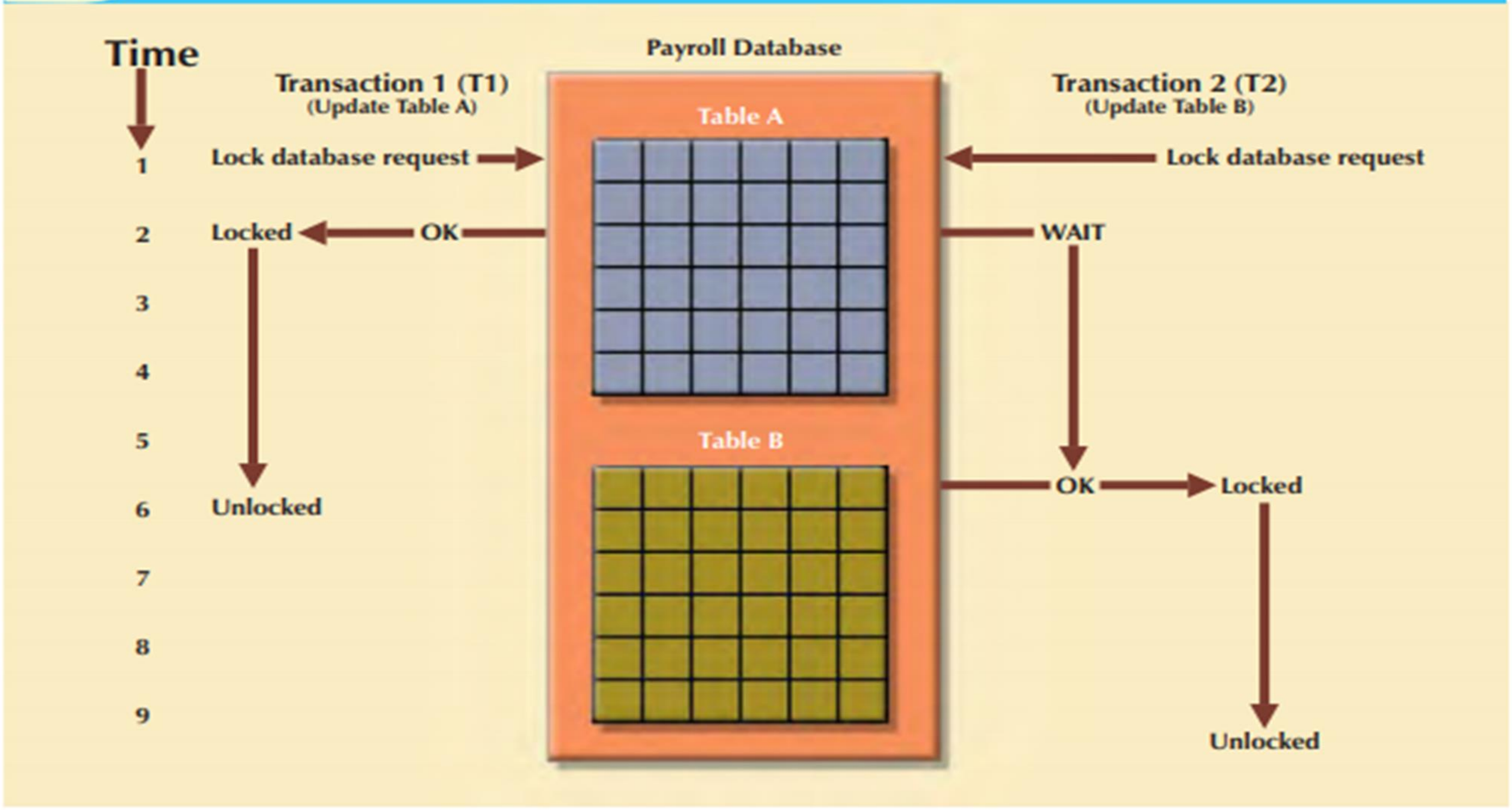
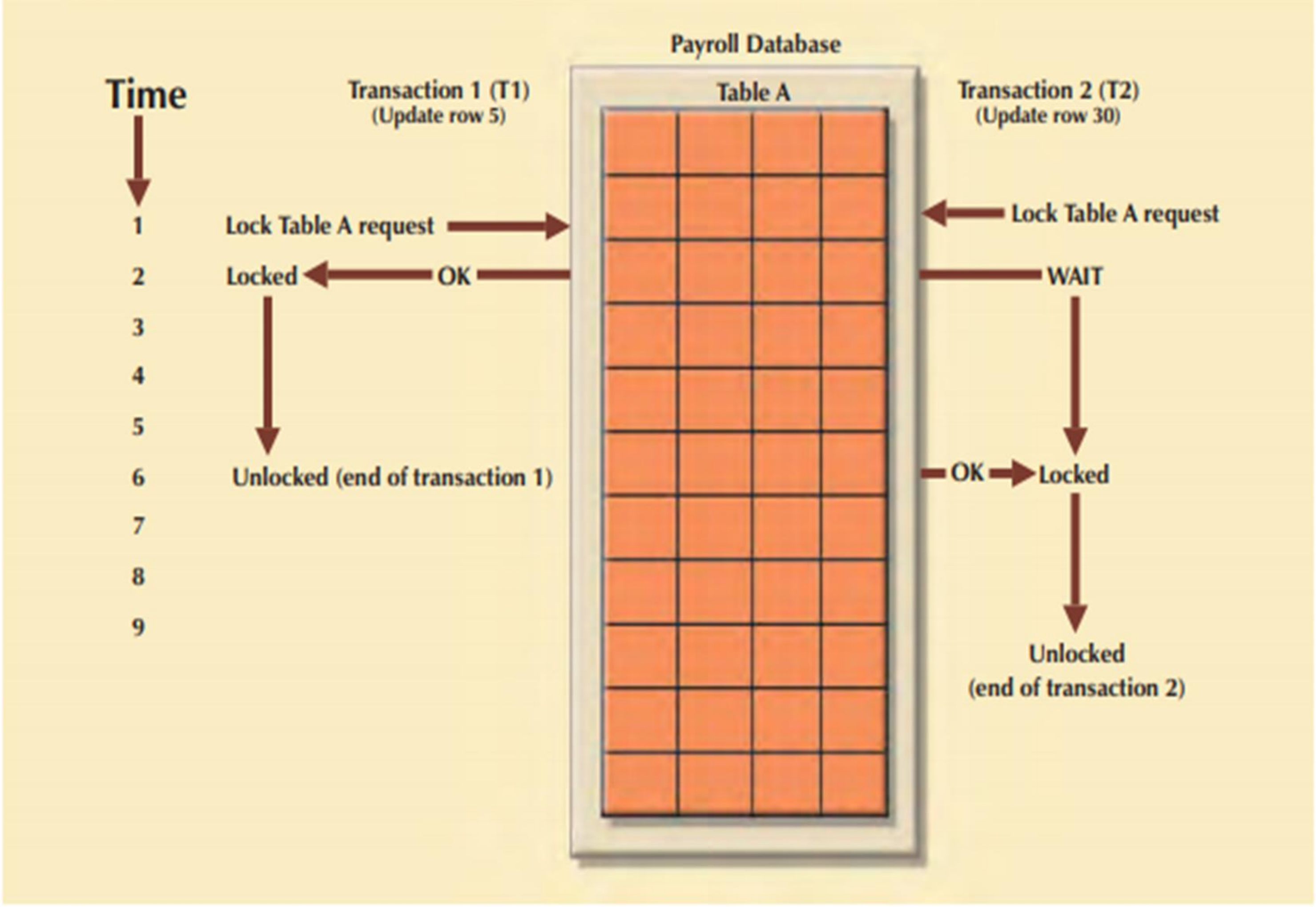


TABLE LEVEL LOCKS

In case of table level locks the entire table (all rows of that table) are locked.
In this case if T1 is using one row of the table it locks the entire table, now if transaction T2 tries to access some other row it will be denied access of the table as it is locked.

FIGURE 10.4 An example of a table-level lock

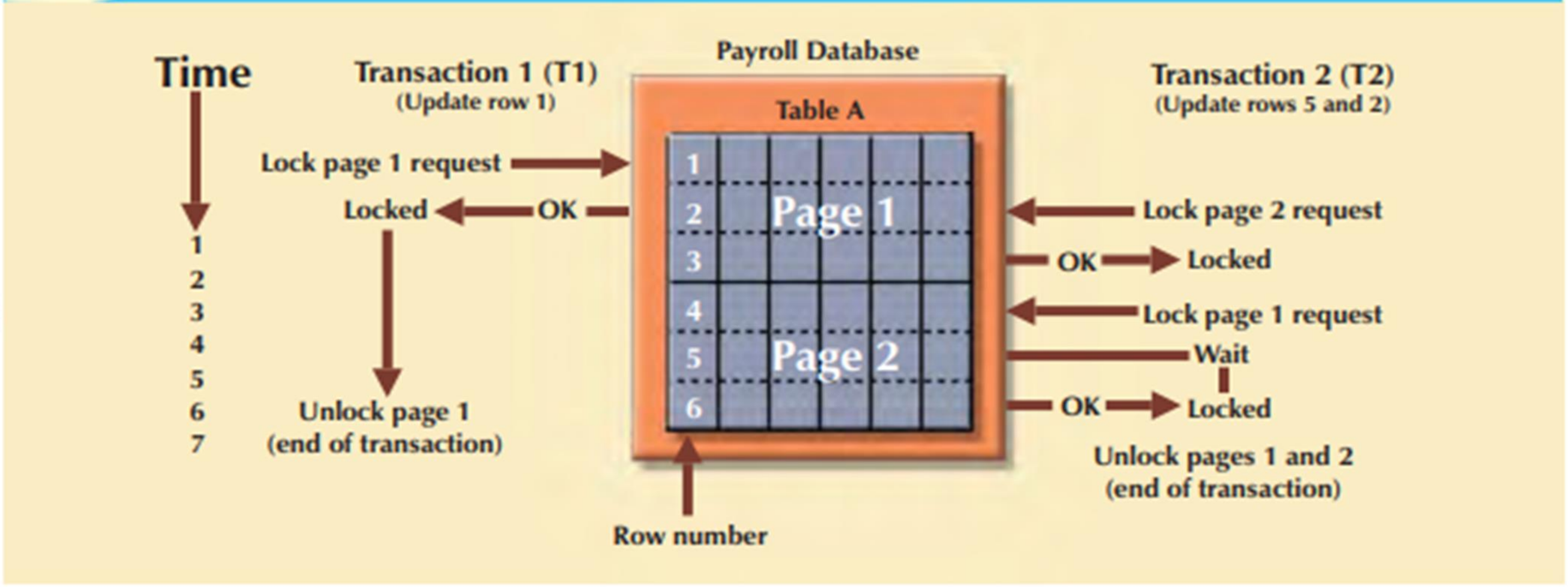


This is better than database level locks but not suitable for multiuser environment if multiple transactions want to access the same table

PAGE LEVEL LOCKS

A page is a block of data. It has a fixed size of say 2k, 4K, etc. One table can have multiple pages, one page can have many rows, or many tables, depending on the size of that page.

FIGURE 10.5 An example of a page-level lock

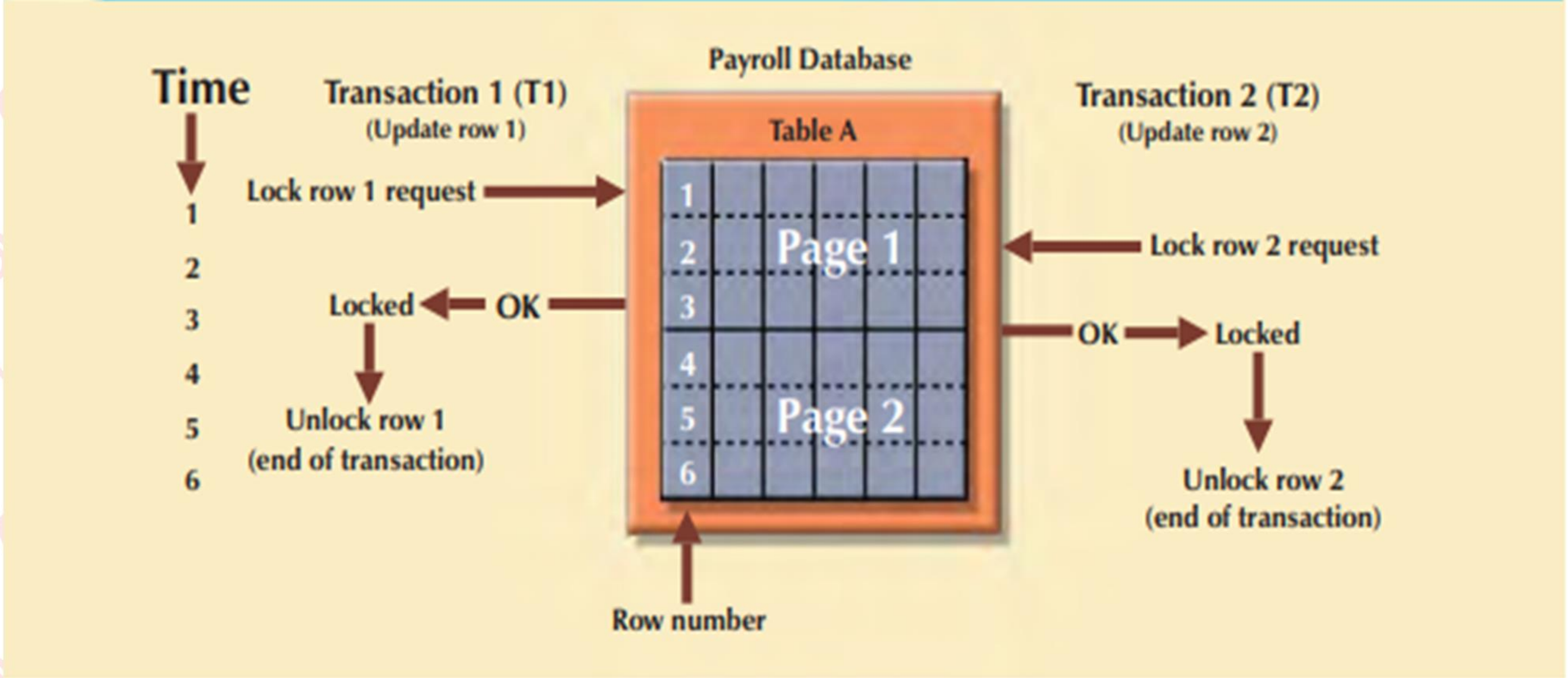


Page level locks are quite efficient in case of multiuser databases.
But Efficiency is further dependent on the page size

ROW LEVEL LOCKS

Here locks are provided to each row of the table. Multiple transactions can thus access multiple rows of the same table at the same time.

FIGURE 10.6 An example of a row-level lock



Improves the access to data as multiple transactions can use different rows of the same table at the same time.
But lot of overhead is involved as locks exists for each row of the table

FIELD LEVEL LOCKS

Here locks are provided for each field (attribute) of the table.

Multiple transaction can use the same row of the table but different fields within that row.

Though it give fast access to data but it is rarely implemented as it requires lot of overhead

LOCK TYPES

Be it any level of lock : Locks are basically of two types>> **BINARY LOCKS**
SHARED LOCKS
EXCLUSIVE LOCKS

BINARY LOCKS

A binary lock has two states: **Locked or Unlocked**

If a row or table or a database is locked by a transaction no other transaction can use that object

When a transaction locks an object, after it finishes its operati

on it has to unlock that object so that some other transaction can use that object.

The lock-unlock operations are managed by the DBMS

Let us see how the problem of LOST UPDATES can be solved by using BINARY LOCKS

1. Serial Execution of Transaction

Time	Transaction	Action	Value
1	T1	Read P_QOH	35
2	T1	P_QOH= 35 + 100	
3	T1	Write P_QOH	135
4	T2	Read P_QOH	135
5	T2	P_QOH = 135-30	
6	T2	Write P_QOH	105

Final value of P_QOH is 105

2. LOST Updates solved using LOCKS

Time	Transaction	Action	Value
1	T1	LOCK PRODUCT	
2	T1	Read P_QOH	35
3	T1	P_QOH = 35 + 100	
4	T1	Write P_QOH	135
5	T1	UNLOCK PRODUCT	
6	T2	LOCK PRODUCT	
7	T2	Read P_QOH	135
8	T2	P_QOH = 135-30	
9	T2	Write P_QOH	105
10	T2	UNLOCK PRODUCT	

Final value of P_QOH is 105

SHARED LOCKS

A shared lock is assigned if all the transactions are “No conflict” or READ opeartions

This means that if T1 and T2 are two transactions who need to access common data item for a READ operation. Since none of the transaction updates the data both transactions can be granted access to the common data item at the same time.
A shared lock is issued if a transaction wants to read data and there is no exclusive lock held on that data item by any other transaction.

EXCLUSIVE LOCKS

An exclusive lock is assigned if a transactions is a “conflict” operation or a WRITE operation.

If two transactions T1 and T2 need to access a common data item to perform a write or update operation then only of the transaction is given   access.
An exclusive lock is issued if a transaction wants to write(update) data and there is no other lock held on that data item by any other transaction.

Problems that can arise due to the use of locks:

1. The transaction is **not serializable.**

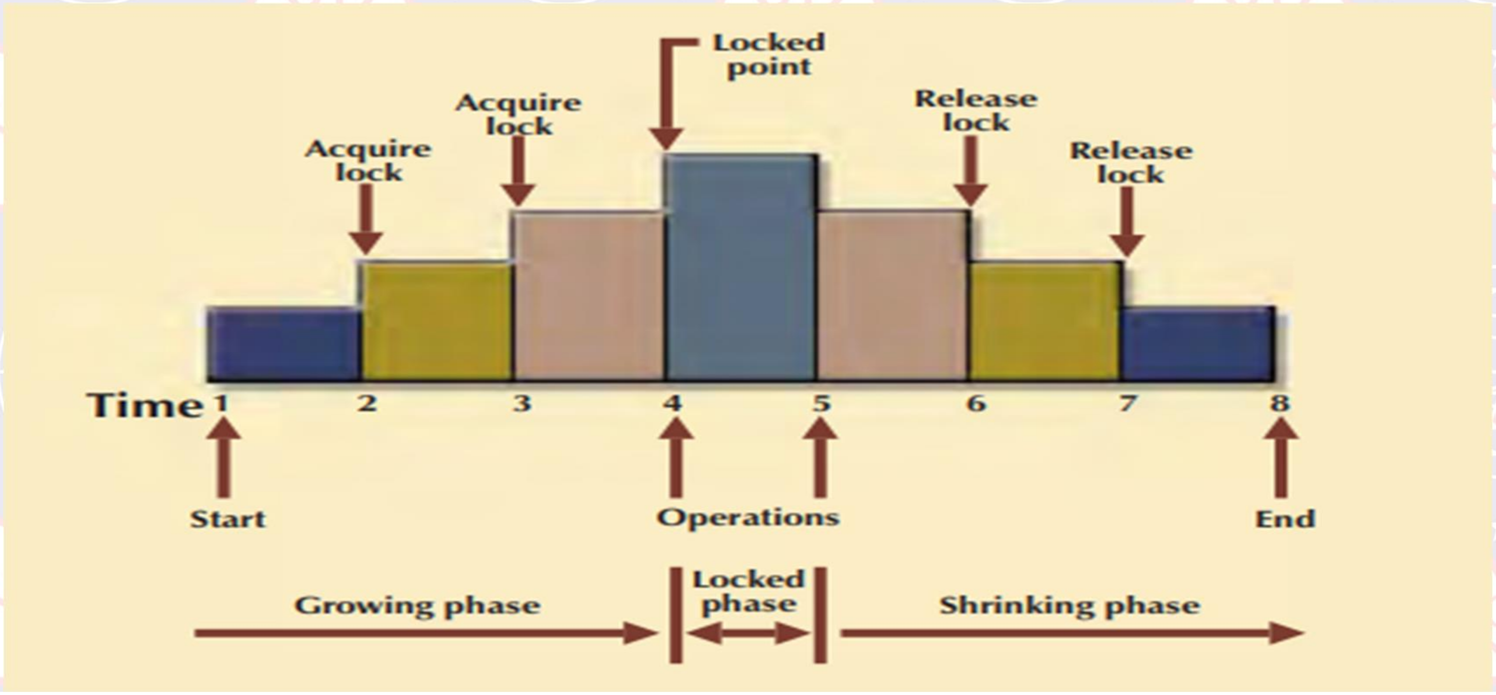
2. **Deadlock**: where two transactions are waiting for each other to unlock data.

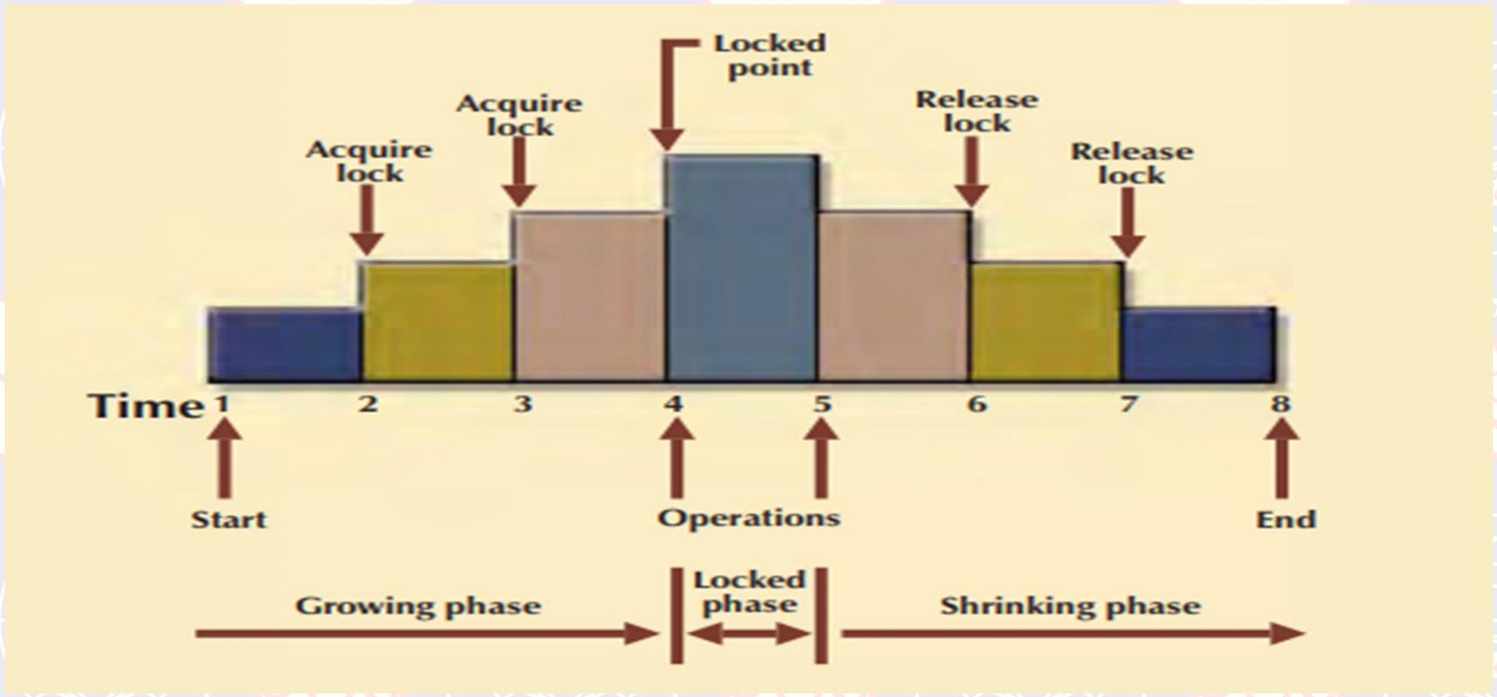
TWO PHASE LOCKING PROTOCOL TO ENSURE SERIALIZABILITY

The two phase locking protocol ensures serializability by defining how the transactions acquire and release locks.

The two phases are :

1. The **GROWING PHASE:** In this phase the transaction acquires all the locks it requires. No UNLOCK (Release of Locks) operation can take place during this phase.
   After all the locks are acquired, the transaction is in the LOCKED state
2. The **SHRINKING PHASE** where the transaction releases all the locks and cannot acquire any new lock.





Rules for two phase locking :

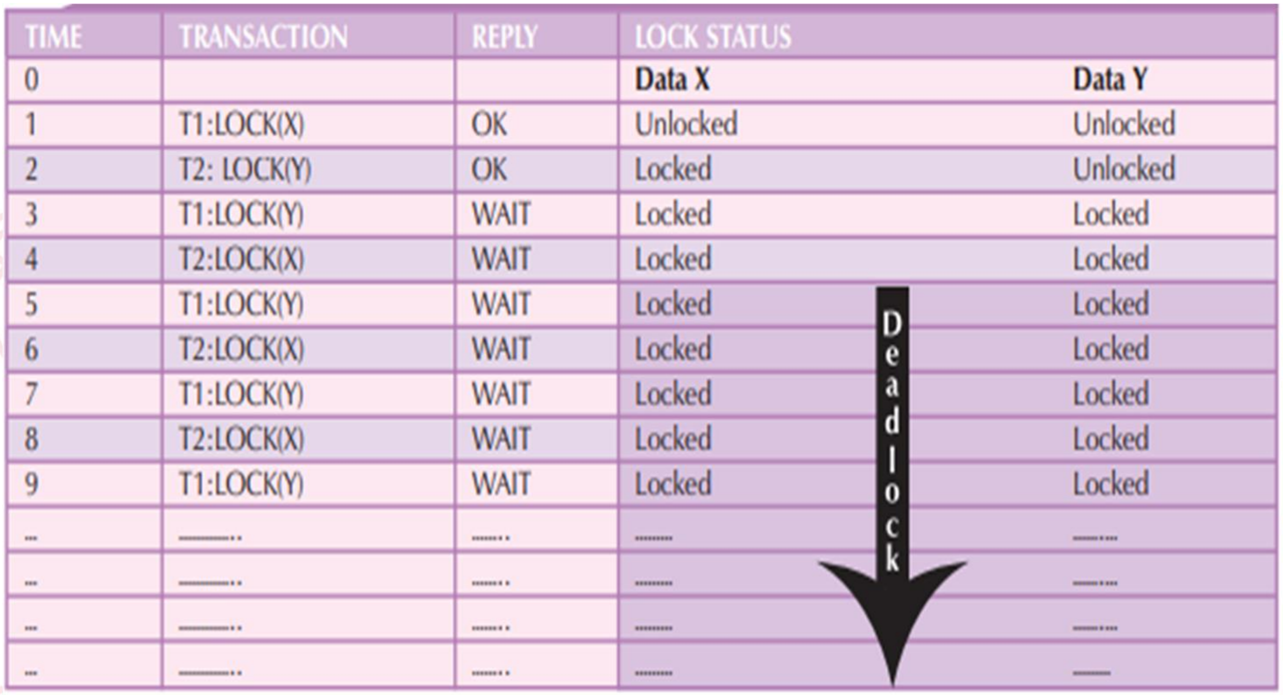
1. Acquire all locks before the transaction actually begins operation
2. Two transactions cannot have conflicting locks. (Conflicting lock: WRITE)
3. In the same transaction there cannot be a UNLOCK operation can come before a LOCK operation.
4. The actual operation or change in data can only take place when a transaction is in its LOCKED PHASE (after acquiring all locks)
5. After the transaction is completes release all locks

Disadvantage : Possibility of deadlock A transaction acquires a lock and waiting to acquire another lock held by some other transaction which is also waiting.

DEADLOCKS

A deadlock occurs when two transactions wait indefinitely for each other to unlock data.

For example: Consider two transactions, T1 and T2
T1 access data items X and Y
T2 access data items Y and X



TIME	TRANSACTION	REPLY	LOCK STATUS	
0			Data X	Data Y
1	T1:LOCK(X)	OK	Unlocked	Unlocked
2	T2: LOCK(Y)	OK	Locked	Unlocked
3	T1:LOCK(Y)	WAIT	Locked	Locked
4	T2:LOCK(X)	WAIT	Locked	Locked
5	T1:LOCK(Y)	WAIT	Locked	Locked
6	T2:LOCK(X)	WAIT	Locked	Locked
7	T1:LOCK(Y)	WAIT	Locked	Locked
8	T2:LOCK(X)	WAIT	Locked	Locked
9	T1:LOCK(Y)	WAIT	Locked	Locked
...	....	....	....	....
...	....	....	....	....
...	....	....	....	....
...	....	....	....	....

T1 and T2 each wait for the other to unlock the required data item.

Such a deadlock is also known as a **deadly embrace.**

The three basic techniques to control deadlocks are:

1. **Deadlock prevention**. A transaction requesting a new lock is aborted when there is the possibility that a deadlock can occur. When the transaction is aborted, all changes made by this transaction are rolled back and all locks are released.
   The transaction is then rescheduled for execution later.

1. **Deadlock detection.** The DBMS periodically tests the database for deadlocks. If a deadlock is found, one of the transactions (known as the "victim") is aborted (rolled back and restarted) and the other transaction continues.

2. **Deadlock avoidance.** The transaction must acquire all of the locks it needs before it starts execution.

The choice of the best deadlock control method to use depends on the database environment.
For example, if the probability of deadlocks is low, deadlock detection is recommended.
However, if the probability of deadlocks is high, deadlock prevention is recommended.
If response time is not high on the system's priority list, deadlock avoidance might be employed.

CONCURRENCY CONTROL WITH TIME STAMPING METHOD

The scheduler schedules the concurrent transactions by the method of **Time stamping.**
In this method a global, unique time stamp value is assigned to each transaction.
The order of execution of the transactions is based on the time stamp value.

Time stamps has two properties: uniqueness and monotonicity.
Uniqueness ensures that two time stamp values cannot be equal
Monotonicity ensures that time stamp values always increase.

All database operations (Read and Write) within the same transaction must have the same time stamp.
For eg:

1. Read your account balance	READ Y	
2. Deduct the amount from your balance	WRITE Y = Y -500	One transaction
3. Write the remaining balance to your account	WRITE Y	
4. Read your friend's account balance	READ F	
5. Add the amount to his account balance	WRITE F = F+500	Same time stamp
6. Write the new updated balance to his account	WRITE F	

The DBMS executes conflicting operations (WRITE) in time stamp order.
This ensures serializability of the transactions.

If two transactions conflict, one is stopped, rolled back, rescheduled, and assigned a new time stamp value.

The disadvantage of the time stamping approach:

- each value stored in the database requires two additional time stamp fields: one for the last time the field was read and one for the last update. This increases memory requirements.

- Lot of processing overheads are also required

- Time stamping method uses a lot of system resources because many transactions might have to be stopped, rescheduled, and restamped

WAIT/DIE AND WOUND/WAIT SCHEME

(For deciding which transaction has to be rolled back and which continues execution)

Consider two conflicting transactions T1 and T2, each with a unique time stamp.

T1 has a time stamp of 11548789 (older transaction) and T2 has a time stamp of 19562545. (new transaction)

There are four possibilities:

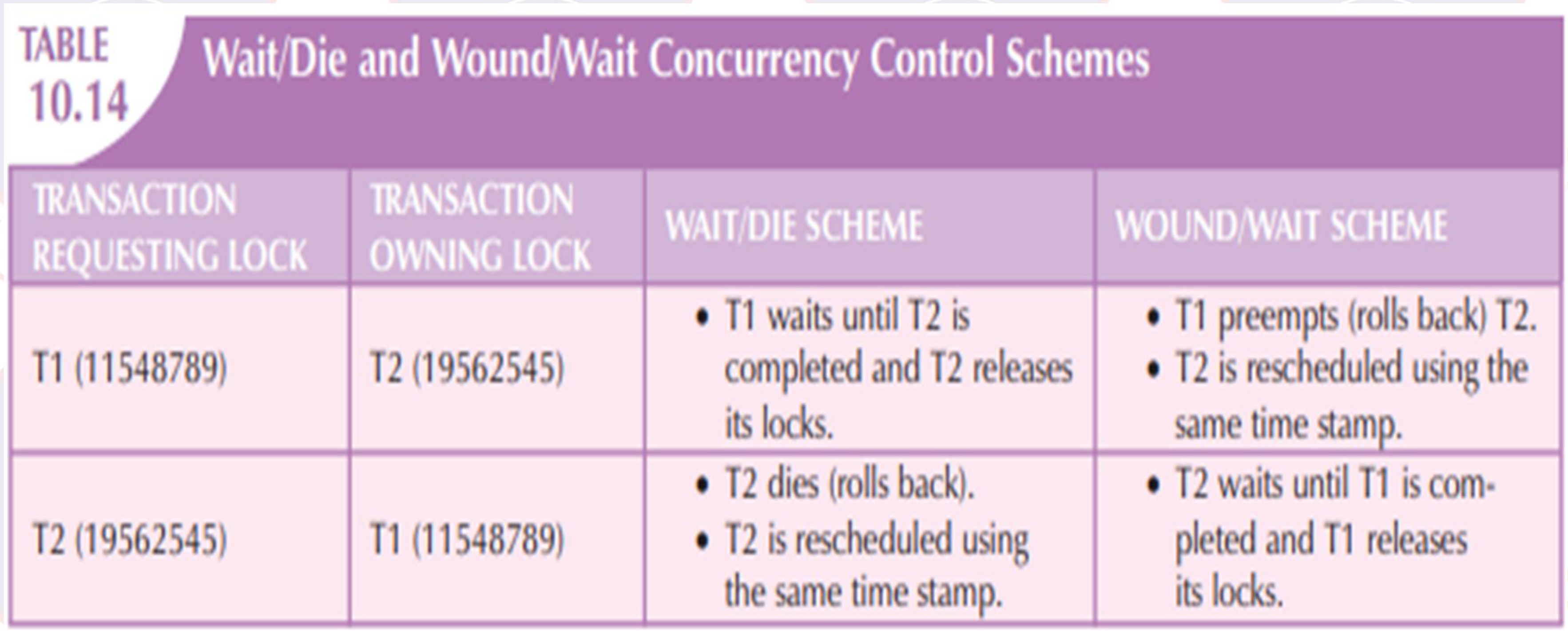


TABLE 10.14 Wait/Die and Wound/Wait Concurrency Control Schemes

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
T1 (11548789)	T2 (19562545)	• T1 waits until T2 is completed and T2 releases its locks.	• T1 preempts (rolls back) T2. • T2 is rescheduled using the same time stamp.
T2 (19562545)	T1 (11548789)	• T2 dies (rolls back). • T2 is rescheduled using the same time stamp.	• T2 waits until T1 is completed and T1 releases its locks.

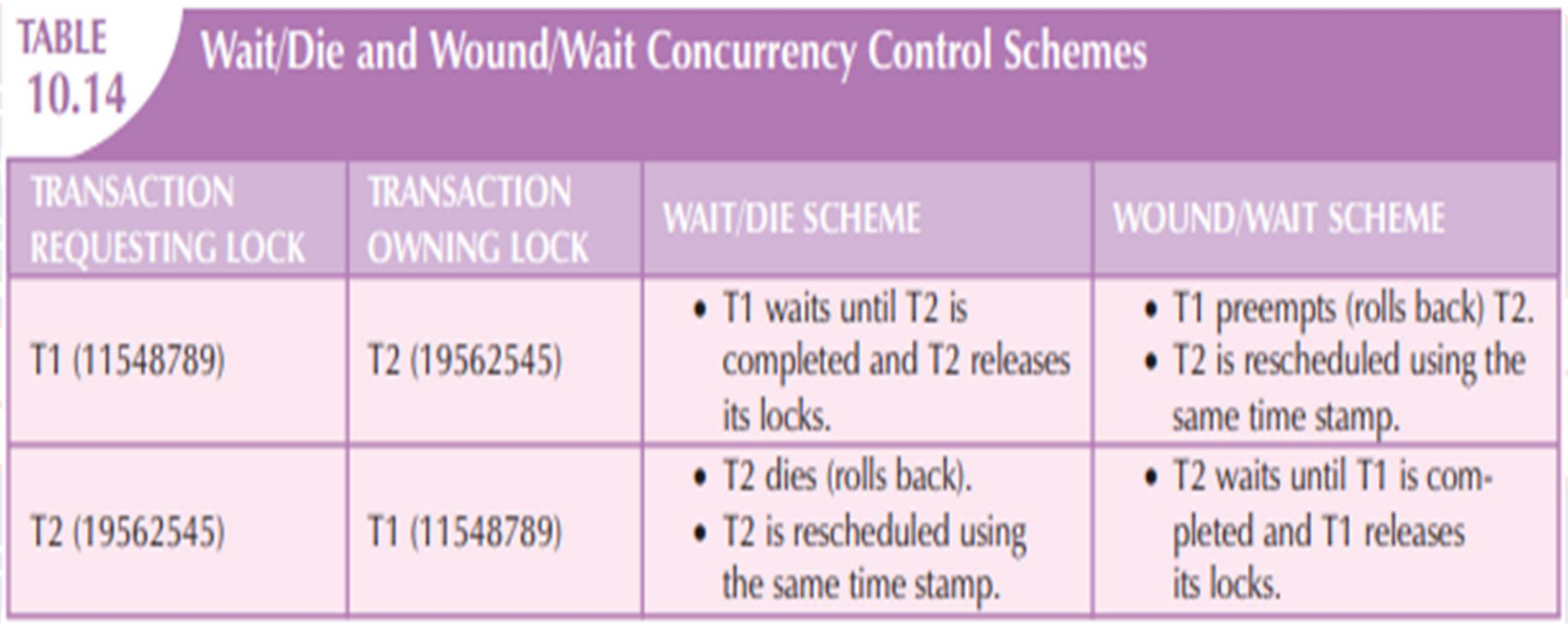


TABLE 10.14 Wait/Die and Wound/Wait Concurrency Control Schemes

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
T1 (11548789)	T2 (19562545)	• T1 waits until T2 is completed and T2 releases its locks.	• T1 preempts (rolls back) T2. • T2 is rescheduled using the same time stamp.
T2 (19562545)	T1 (11548789)	• T2 dies (rolls back). • T2 is rescheduled using the same time stamp.	• T2 waits until T1 is completed and T1 releases its locks.

Using the wait/die scheme:

- If the transaction requesting the lock is the older of the two transactions, it will wait until the other transaction is completed and the locks are released.
- If the transaction requesting the lock is the younger of the two transactions, it will die (roll back) and is rescheduled using the same time stamp. **In the wait/die scheme, the older transaction waits for the younger to complete and release its locks.**

Using wound/wait scheme:

- If the transaction requesting the lock is the older , it will preempt (wound) the younger transaction (by rolling it back). T1 preempts T2 when T1 rolls back T2. T2 is rescheduled using same time stamp.
- If the transaction requesting the lock is the younger transactions, it will wait until the other transaction is completed and locks are released.**In the wound/wait scheme, the older transaction rolls back the younger transaction and reschedules it**

CONCURRENCY CONTROL WITH OPTIMISTIC METHOD

The optimistic approach requires neither locking nor time stamping techniques.

It is based on an optimistic approach assuming that the majority of the database operations do not conflict.
Using an optimistic approach :
A transaction is executed without restrictions until it is committed.
Each transaction moves through two or three phases, referred to as **read**, **validation**, and **write**.
During the read phase, the transaction reads the database, executes the needed computations, and makes the updates to a temporary copy of the database values which is not accessed by the remaining transactions.
During the validation phase, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to the write phase. If the validation test is negative, the transaction is restarted and the changes are discarded.
During the write phase, the changes are permanently applied to the database.

This approach works well if most transactions are read transactions.

DATABASE RECOVERY MANAGEMENT

Need:
Database Recovery management is required whenever a transaction has to be aborted. At that point of time the transaction has to be rolled back. This means that the all the operations have top undone and the **database has to be brought from a given state (which is usually a an inconsistent state) to a previous consistent state.**

The need for database recovery management can also be due to certain critical events like:

1. Hardware/Software failures : This includes problem like disk crash, problem in motherboard, OS problems, loss of data etc.

2. Human caused incidents: This can be intentional or unintentional. Unintentional incidents can be like deleting data accidently, pressing the wrong key , or accidental shut down. Intentional incidents can be like: hackers trying to have unauthorized access to data resources and virus attacks caused by employees for all wrong reasons.

3. Natural disasters like fires, earthquakes, floods, and power failure

Techniques for Transaction Recovery

A Transaction Log can be used for recovering a database from an inconsistent state to the previous consistent state.

The protocols for the recovery process are :

- The **write-ahead-log protocol** ensures that transaction logs are always written before any data is actually updated. This ensures that, in case of a failure, the database can be recovered to a previous consistent state, using the data in the transaction log.
- **Redundant transaction logs** or many copies of the transaction log are maintained to ensure that even if one copy is damaged the DBMS will be able to recover data anyhow.
- Database **buffers** or temporary storage areas of RAM are used to speed up operations. To improve processing time, the DBMS reads the data from the physical disk and stores a copy of it on a “buffer”. When a transaction updates data, it actually updates the copy of the data in the buffer which is much faster than accessing the physical disk every time. Later on, with a single operation the data from the buffers are written to a physical disk
- Database **checkpoints** are created Checkpoints is the time when DBMS writes all of its updated buffers to disk. While this is happening, the DBMS does not execute any other requests. This operation is registered in the transaction log

To bring the database to a consistent state after a failure the recovery technique uses two methods :

1. **Deferred Write or deferred Update Technique :** In this technique the transaction operations do not update the physical database immediately. Only the transaction log is updated. The physical database is updated only when the transaction performs a Commit. The advantage of this is that if the transaction is aborted before performing commit there is no need of rollback because the changes were never made to the physical database.

2. **Write Through or Immediate Update Technique :**

   In this technique the database is immediately updated by transaction operations even before the transaction reaches its commit point. If the transaction aborts before it reaches its commit point, a ROLLBACK or undo operation needs to be done to restore the database to a consistent state. In that case, the ROLLBACK operation will use the 'before' values of transaction log.