Module 3

Syllabus

Transaction Management and Concurrency Control:-

Transaction: Evaluating Transaction Results, Transaction Properties, Transaction Management with SQL, The Transaction Log –Concurrency Control: Lost Updates, Uncommitted Data, Inconsistent Retrievals, The Scheduler Concurrency Control with Locking Methods: Lock Granularity, Lock Types, Two Phase Locking to Ensure Serializability, Deadlocks – Concurrency Control with Timestamping Methods: Wait/Die and Wait/Wound Schemes – Concurrency Control with Optimistic Methods Database Recovery Management: Transaction Recovery

Transaction

- In database terms, a **transaction is any action that** reads from and/or writes to a database.
- A transaction may consist of a simple SELECT statement to generate a list of table contents; it may consist of a series of related UPDATE statements to change the values of attributes in various tables; it may consist of a series of INSERT statements to add rows to one or more tables, or it may consist of a combination of SELECT, UPDATE, and INSERT statement
- A transaction is a *logical unit* of work that must be entirely completed or entirely aborted; no intermediate states are acceptable
- All of the SQL statements in the transaction must be completed successfully. If any of the SQL statements fail, the entire transaction is rolled back to the original database state that existed before the transaction started

Transaction

- A successful transaction changes the database from one consistent state to another.
- A **consistent database state is one in which all data integrity constraints are satisfied.**
- To ensure consistency of the database, every transaction must begin with the database in a known consistent state. If the database is not in a consistent state, the transaction will yield an inconsistent database that violates its integrity and business rules.

Transaction Properties

- Each individual transaction must display *atomicity, consistency, isolation, and durability. These properties are* sometimes referred to as the ACID test.
- In addition, when executing multiple transactions, the DBMS must schedule the concurrent execution of the transaction’s operations. The schedule of such transaction’s operations must exhibit the property of *serializability*

1. **Atomicity** requires that *all operations (SQL requests) of a transaction be completed; if not, the transaction is* aborted.

- If a transaction T1 has four SQL requests, all four requests must be successfully completed; otherwise, the entire transaction is aborted. In other words, a transaction is treated as a single, indivisible, logical unit of work.

2. **Consistency** indicates the permanence of the database’s consistent state. A transaction takes a database from one consistent state to another consistent state. When a transaction is completed, the database must be in a consistent state; if any of the transaction parts violates an integrity constraint, the entire transaction is aborted.

Transaction Properties

3. **Isolation** means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.

- In other words, if a transaction T1 is being executed and is using the data item X, that data item cannot be accessed by any other transaction (T2 ... Tn) until T1 ends.
- This property is particularly useful in multiuser database environments because several users can access and update the database at the same time.

4. **Durability** ensures that once transaction changes are done (committed), they cannot be undone or lost, even in the event of a system failure.

5. **Serializability** ensures that the schedule for the concurrent execution of the transactions yields consistent results.

- This property is important in multiuser and distributed databases, where multiple transactions are likely to be executed concurrently.
- Naturally, if only a single transaction is executed, serializability is not an issue

Transaction Management with SQL

- Transaction support is provided by two SQL statements: COMMIT and ROLLBACK
- when a transaction sequence is initiated by a user or an application program, the sequence must continue through all succeeding SQL statements until one of the following four events occurs:

- A COMMIT statement is reached, in which case all changes are permanently recorded within the database. The COMMIT statement automatically ends the SQL transaction.
- A ROLLBACK statement is reached, in which case all changes are aborted and the database is rolled back to its previous consistent state.
- The end of a program is successfully reached, in which case all changes are permanently recorded within the database. This action is equivalent to COMMIT.
- The program is abnormally terminated, in which case the changes made in the database are aborted and the database is rolled back to its previous consistent state. This action is equivalent to ROLLBACK.

Transaction Management with SQL

Example

```
UPDATE PRODUCT
SET PROD_QOH = PROD_QOH – 2
WHERE PROD_CODE = '1558-QW1';
UPDATE CUSTOMER
SET CUST_BALANCE = CUST_BALANCE + 87.98
WHERE CUST_NUMBER = '10011';
COMMIT;
```

- Actually, the COMMIT statement used in that example is not necessary if the UPDATE statement is the application’s last action and the application terminates normally
- A transaction begins implicitly when the first SQL statement is encountered.
- BEGIN _TRANSACTION and END_TRANSACTION

Transaction Log

- A DBMS uses a **transaction log to keep track of all transactions that update the database.**
- **The information stored** in this log is used by the DBMS for a recovery requirement triggered by a ROLLBACK statement, a program’s abnormal termination, or a system failure such as a network discrepancy or a disk crash.
- Some RDBMSs use the transaction log to recover a database *forward to a currently consistent state.*
- *After a server failure, for example, Oracle* automatically rolls back uncommitted transactions and rolls forward transactions that were committed but not yet written to the physical database.
- This behavior is required for transactional correctness and is typical of any transactional DBMS.

Transaction Log

- While the DBMS executes transactions that modify the database, it also automatically updates the transaction log. The transaction log stores:
- A record for the beginning of the transaction.
- For each transaction component (SQL statement):

  - The type of operation being performed (update, delete, insert).

  - The names of the objects affected by the transaction (the name of the table).

  - The "before" and "after" values for the fields being updated.

  - Pointers to the previous and next transaction log entries for the same transaction.
- The ending (COMMIT) of the transaction.

Transaction Log

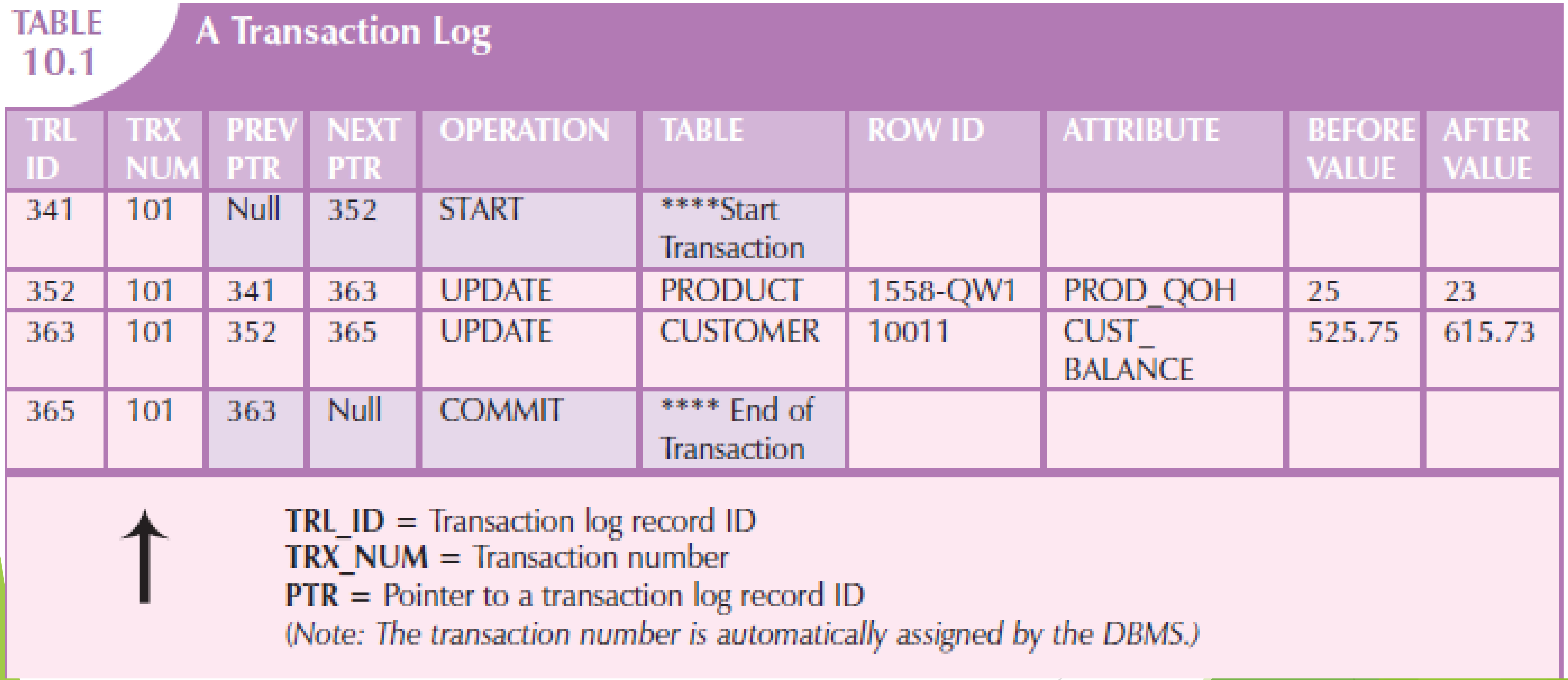


TABLE 10.1 A Transaction Log

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.)

Transaction Log

- Table illustrates a simplified transaction log that reflects a basic transaction composed of two SQL UPDATE statements.
- If a system failure occurs, the DBMS will examine the transaction log for all uncommitted or incomplete transactions and restore (ROLLBACK) the database to its previous state on the basis of that information.
- When the recovery process is completed, the DBMS will write in the log all committed transactions that were not physically written to the database before the failure occurred.
- If a ROLLBACK is issued before the termination of a transaction, the DBMS will restore the database only for that particular transaction, rather than for all transactions, to maintain the *durability of the previous transactions*.
- *In other* words, committed transactions are not rolled back.

CONCURRENCY CONTROL

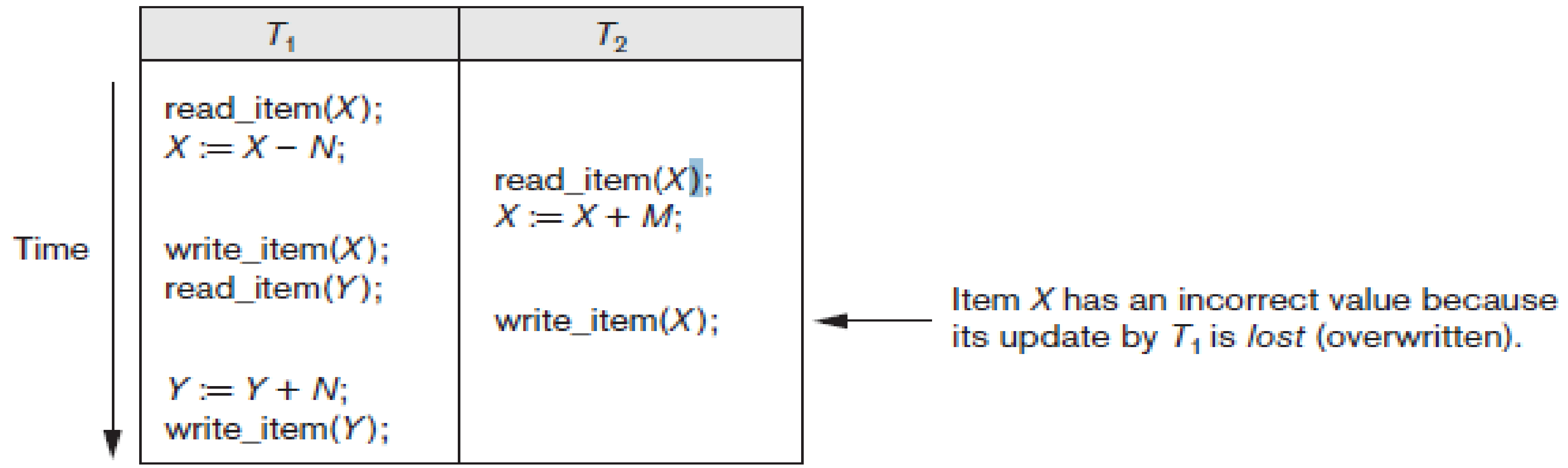
- The coordination of the simultaneous execution of transactions in a multiuser database system is known as **concurrency control.**
- **The objective of concurrency control is to ensure the serializability of transactions in a** multiuser database environment.
- Concurrency control is important because the simultaneous execution of transactions over a shared database can create several data integrity and consistency problems.
- The three main problems are lost updates, uncommitted data, and inconsistent retrievals.

CONCURRENCY CONTROL

Lost Updates

- The lost update problem occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the updates is lost (overwritten by the other transaction).
- Suppose that transactions *T1 and T2 are submitted at* approximately the same time, and suppose that their operations are interleaved; then the final value of item *X is incorrect because T2 reads the* value of *X before T1 changes it in the database, and hence the updated value resulting* from *T1 is lost.*

(a)



Time ↓	T_1	T_2	
	read_item(X); $X := X - N$;		
		read_item(X); $X := X + M$;	
	write_item(X); read_item(Y);		
		write_item(X);	← Item X has an incorrect value because its update by T_1 is *lost* (overwritten).
	$Y := Y + N$; write_item(Y);		

CONCURRENCY CONTROL

Lost Updates

- For example, if X = 80 at the start (originally there were 80 reservations on the flight), N = 5 (T1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and M = 4 (T2 reserves 4 seats on X), the final result should be X = 79. However, in the interleaving of operations shown in Figure 20.3(a), it is X = 84 because the update in T1 that removed the five seats from X was lost.

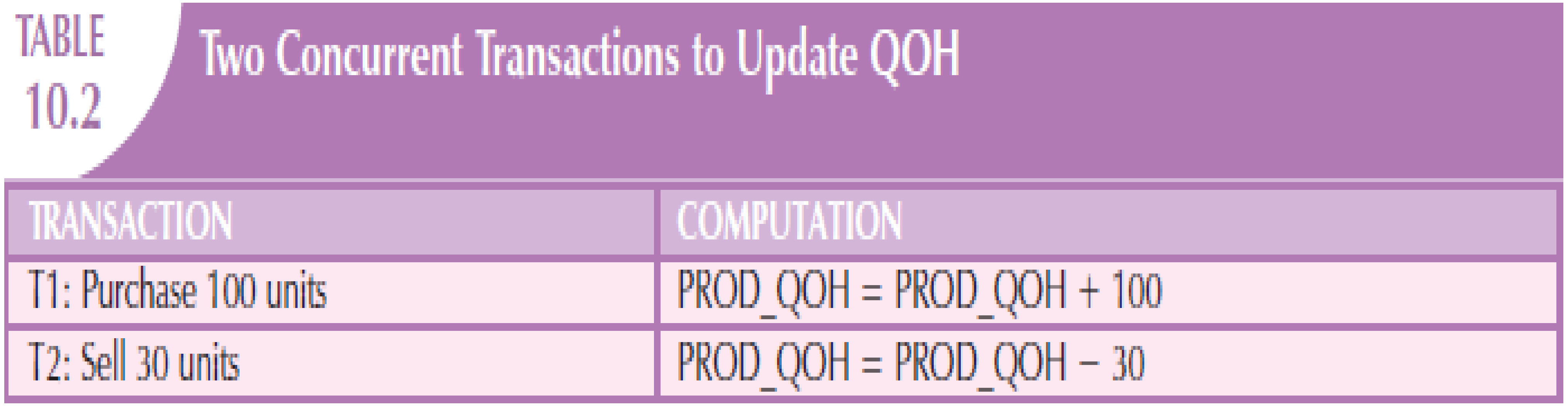


TABLE 10.2 Two Concurrent Transactions to Update QOH

TRANSACTION	COMPUTATION
T1: Purchase 100 units	PROD_QOH = PROD_QOH + 100
T2: Sell 30 units	PROD_QOH = PROD_QOH − 30

CONCURRENCY CONTROL

Lost Updates

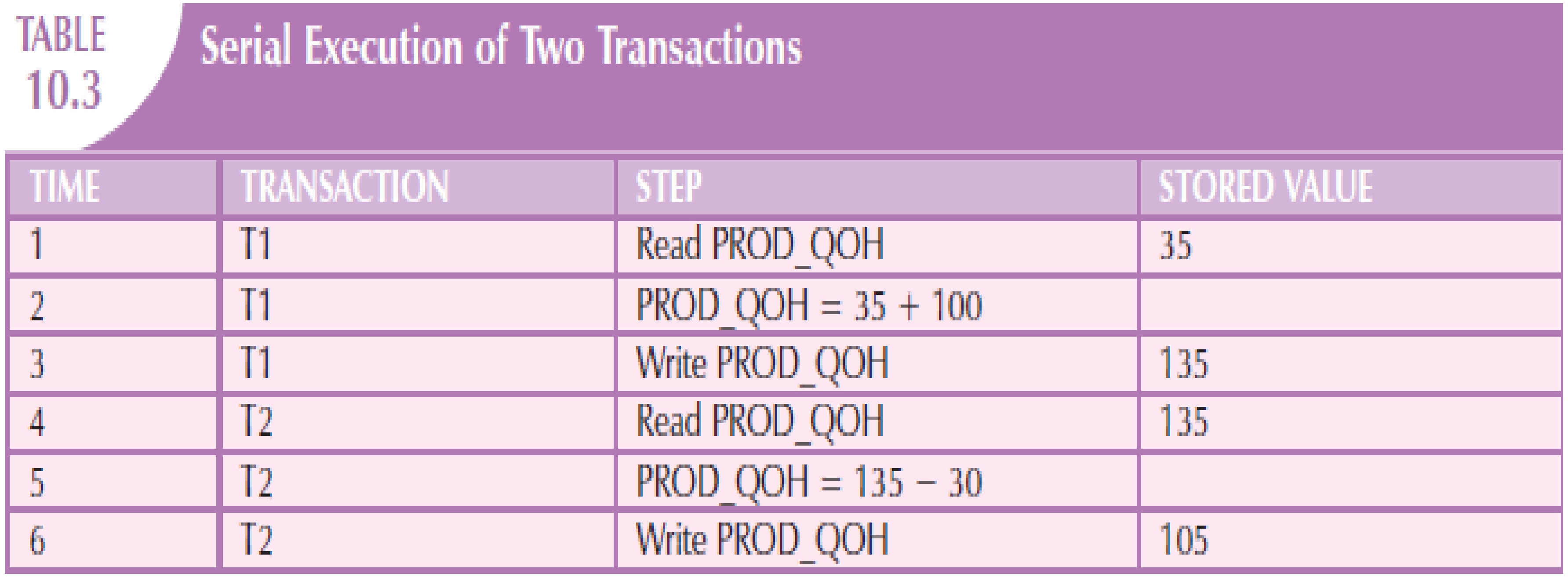


TABLE 10.3 Serial Execution of Two Transactions

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T1	PROD_QOH = 35 + 100	
3	T1	Write PROD_QOH	135
4	T2	Read PROD_QOH	135
5	T2	PROD_QOH = 135 − 30	
6	T2	Write PROD_QOH	105

- But suppose that a transaction is able to read a product’s PROD_QOH value from the table before a previous transaction (using the same product) has been committed

CONCURRENCY CONTROL

Lost Updates

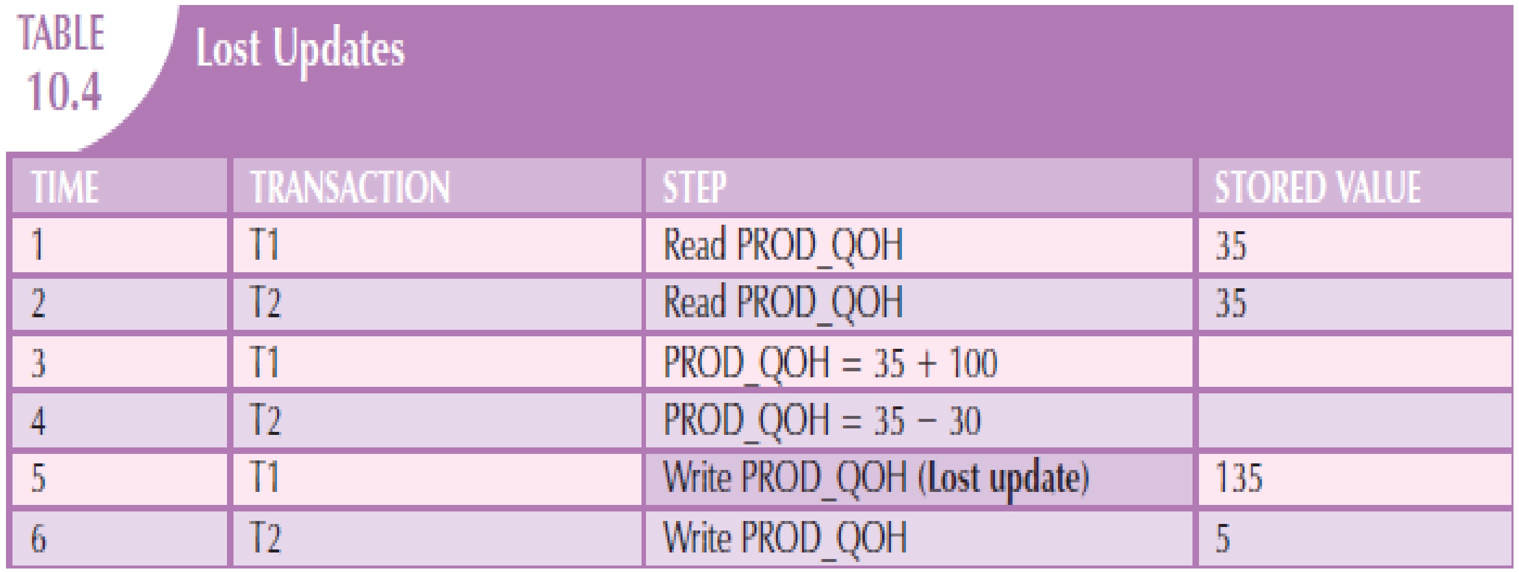


TABLE 10.4 Lost Updates

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T2	Read PROD_QOH	35
3	T1	PROD_QOH = 35 + 100	
4	T2	PROD_QOH = 35 − 30	
5	T1	Write PROD_QOH (**Lost update**)	135
6	T2	Write PROD_QOH	5

- Note that the first transaction (T1) has not yet been committed when the second transaction(T2) is executed.
- Therefore, T2 still operates on the value 35, and its subtraction yields 5 in memory. In the meantime,T1 writes the value 135 to disk, which is promptly overwritten by T2. In short, the addition of 100 units is “lost” during the process.

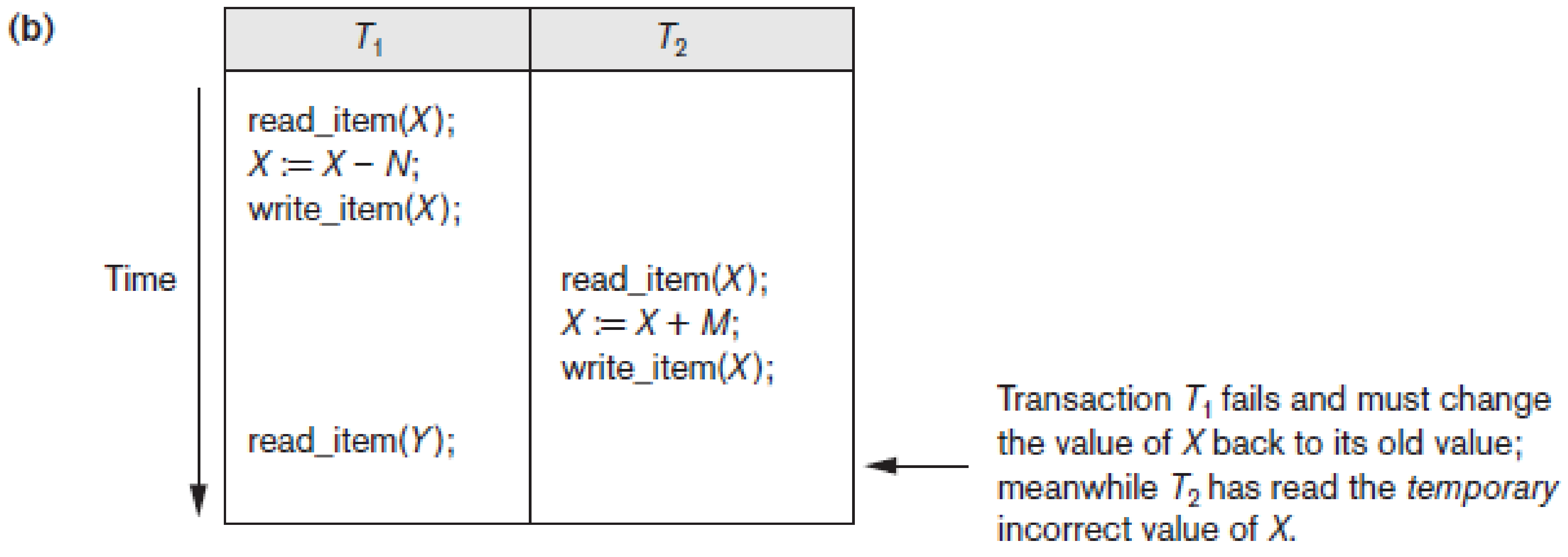
CONCURRENCY CONTROL

Uncommitted Data

- The phenomenon of uncommitted data occurs when two transactions, T1 and T2, are executed concurrently and the first transaction (T1) is rolled back after the second transaction (T2) has already accessed the uncommitted data—thus violating the isolation property of transactions
- T1 updates item X and then fails before completion, so the system must roll back X to its original value. Before it can do so, however, transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T1.
- The value of item X that is read by T2 is called dirty data because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

CONCURRENCY CONTROL

Uncommitted Data



(b)

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

CONCURRENCY CONTROL

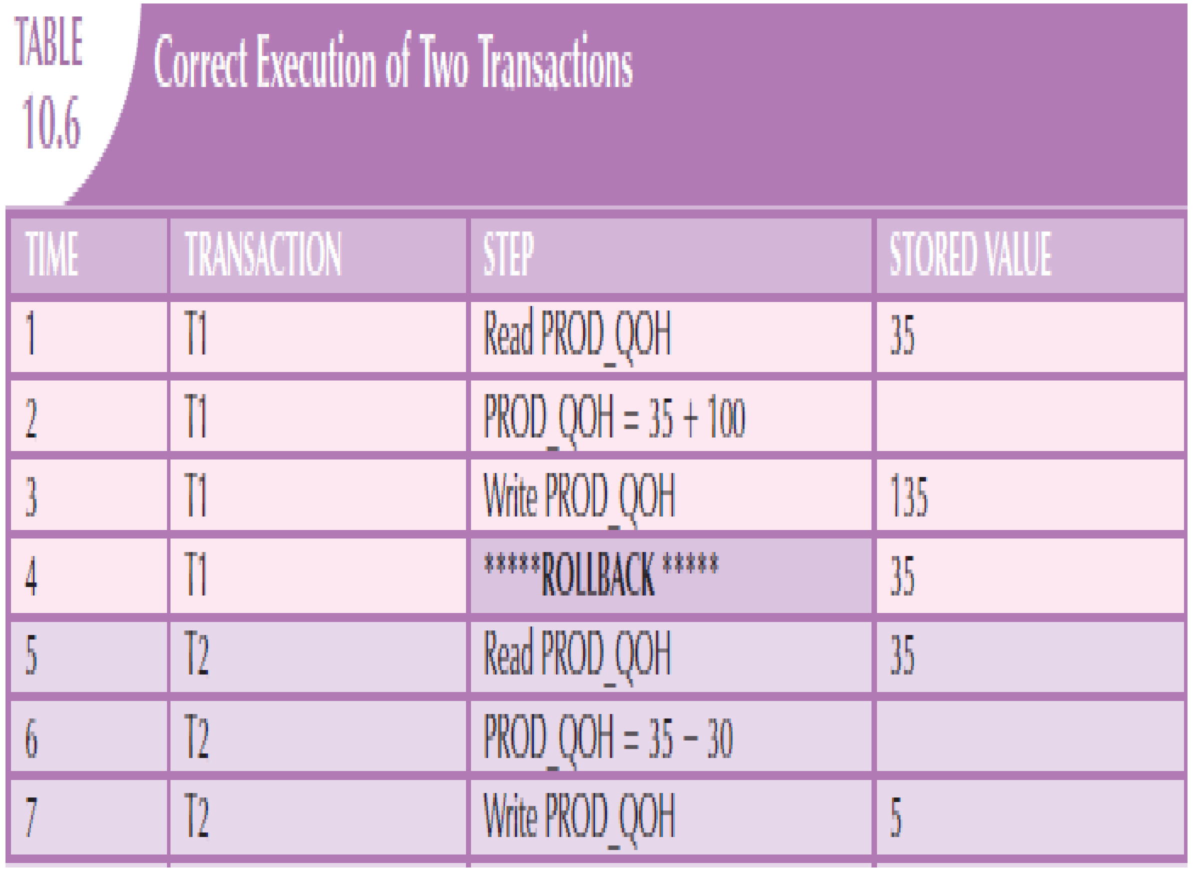


TABLE 10.6 Correct Execution of Two Transactions

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T1	PROD_QOH = 35 + 100	
3	T1	Write PROD_QOH	135
4	T1	*****ROLLBACK *****	35
5	T2	Read PROD_QOH	35
6	T2	PROD_QOH = 35 − 30	
7	T2	Write PROD_QOH	5

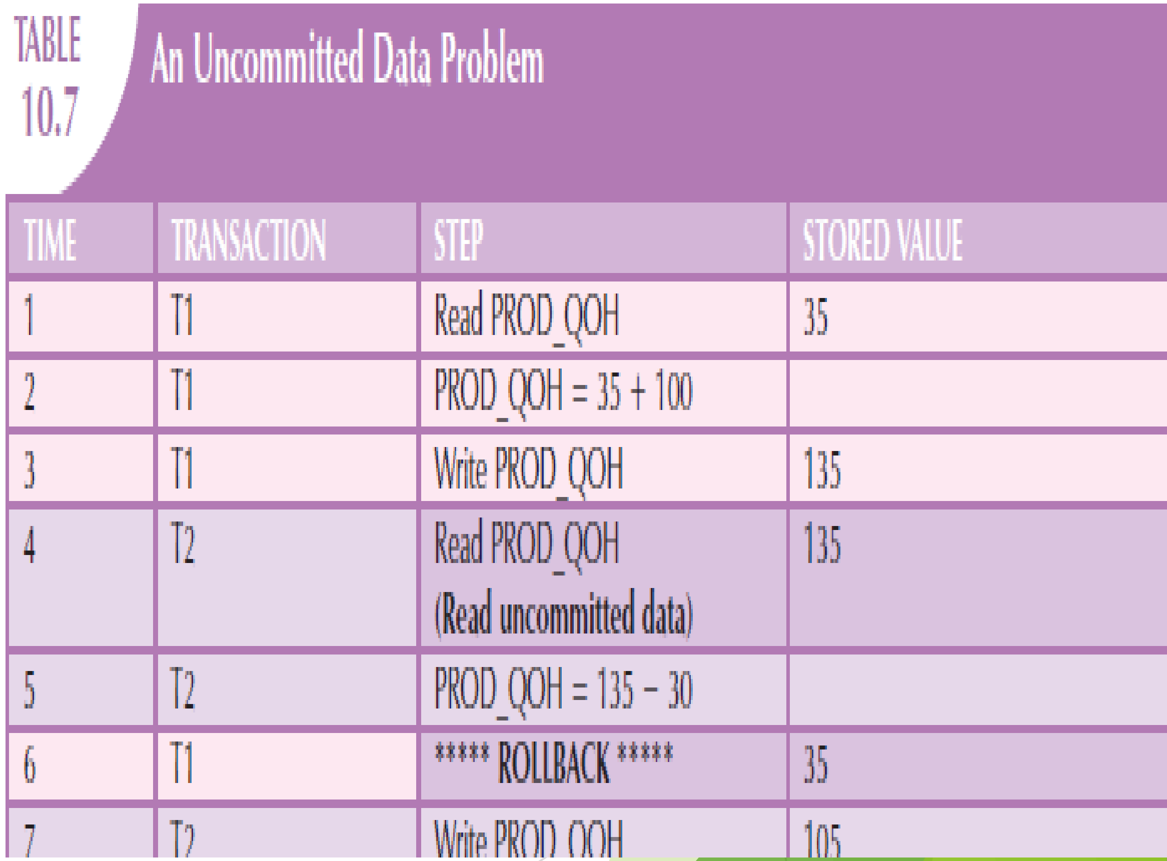


TABLE 10.7 An Uncommitted Data Problem

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T1	PROD_QOH = 35 + 100	
3	T1	Write PROD_QOH	135
4	T2	Read PROD_QOH (Read uncommitted data)	135
5	T2	PROD_QOH = 135 − 30	
6	T1	***** ROLLBACK *****	35
7	T2	Write PROD_QOH	105

CONCURRENCY CONTROL

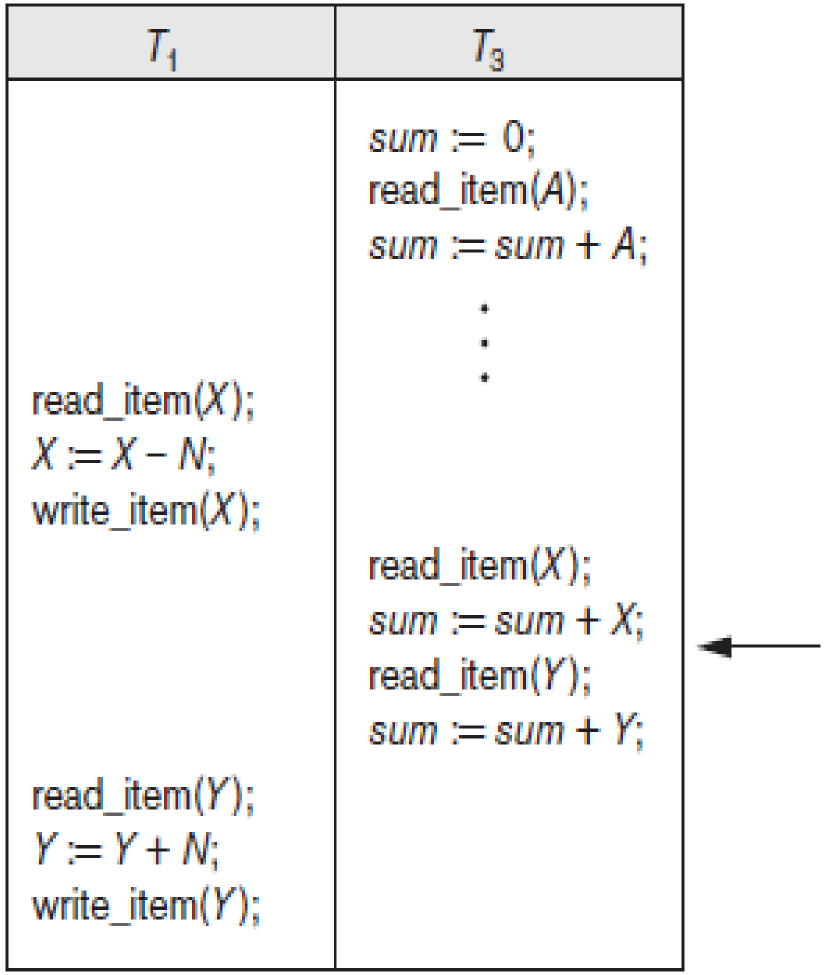
Inconsistent Retrievals

- Inconsistent retrievals occur when a transaction accesses data before and after another transaction(s) finish working with such data.
- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.
- For example, suppose that a transaction T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing.
- If the interleaving of operations shown in occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

CONCURRENCY CONTROL

Inconsistent Retrieval

(c)



T_1	T_3
	$sum := 0$; read_item(A); $sum := sum + A$; ⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

CONCURRENCY CONTROL

The Scheduler

- Severe problems can arise when two or more concurrent transactions are executed
- A database transaction involves a series of database I/O operations that take the database from one consistent state to another
- Database consistency can be ensured only before and after the execution of transactions.
- A database always moves through an unavoidable temporary state of inconsistency during a transaction's execution if such transaction updates multiple tables/rows
- The scheduler is a special DBMS process that establishes the order in which the operations within concurrent transactions are executed.
- The scheduler interleaves the execution of database operations to ensure serializability and isolation of transactions.
- To determine the appropriate order, the scheduler bases its actions on concurrency control algorithms, such as locking or time stamping method

CONCURRENCY CONTROL

The Scheduler

- The DBMS determines what transactions are serializable and proceeds to interleave the execution of the transaction’s operations. Generally, transactions that are not serializable are executed on a first-come, first-served basis by the DBMS.
- The scheduler’s main job is to create a serializable schedule of a transaction’s operations.
- A **serializable schedule is a schedule of a transaction’s operations** in which the interleaved execution of the transactions (T1, T2, T3, etc.) yields the same results as if the transactions were executed in serial order (one after another).

CONCURRENCY CONTROL

The Scheduler

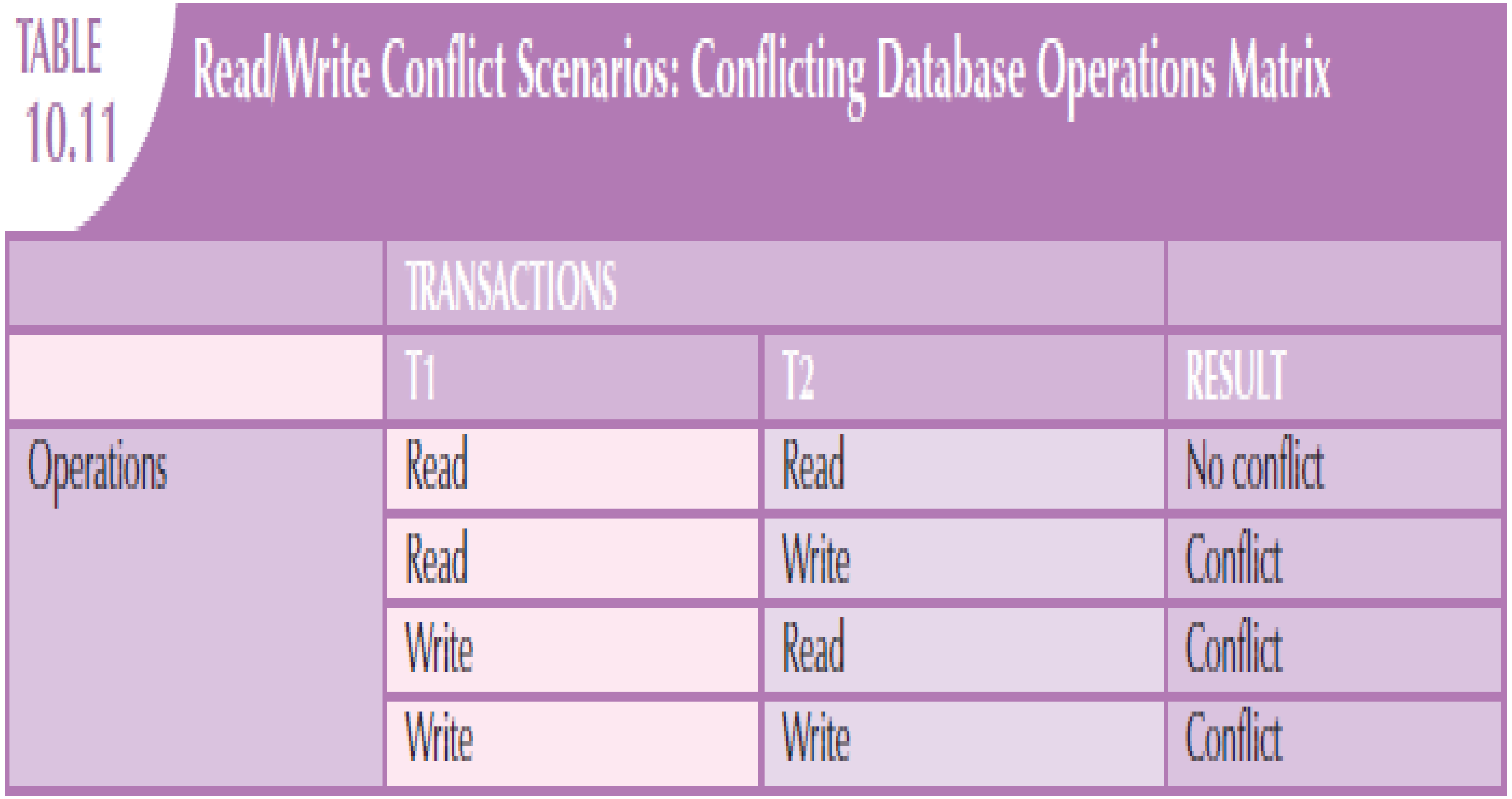


TABLE 10.11 Read/Write Conflict Scenarios: Conflicting Database Operations Matrix

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

CONCURRENCY CONTROL

CONCURRENCY CONTROL WITH LOCKING METHODS

- A lock guarantees exclusive use of a data item to a current transaction. In other words, transaction T2 does not have access to a data item that is currently being used by transaction T1.
- A transaction acquires a lock prior to data access; the lock is released (unlocked) when the transaction is completed so that another transaction can lock the data item for its exclusive use.
- This series of locking actions assumes that there is a likelihood of concurrent transactions attempting to manipulate the same data item at the same time. The use of locks based on the assumption that conflict between transactions is likely to occur is often referred to as **pessimistic locking.**
- Most multiuser DBMSs automatically initiate and enforce locking procedures. All lock information is managed by a **lock manager, which is responsible for assigning and policing the locks used by the transactions.**

CONCURRENCY CONTROL

CONCURRENCY CONTROL WITH LOCKING METHODS

a)Lock Granularity

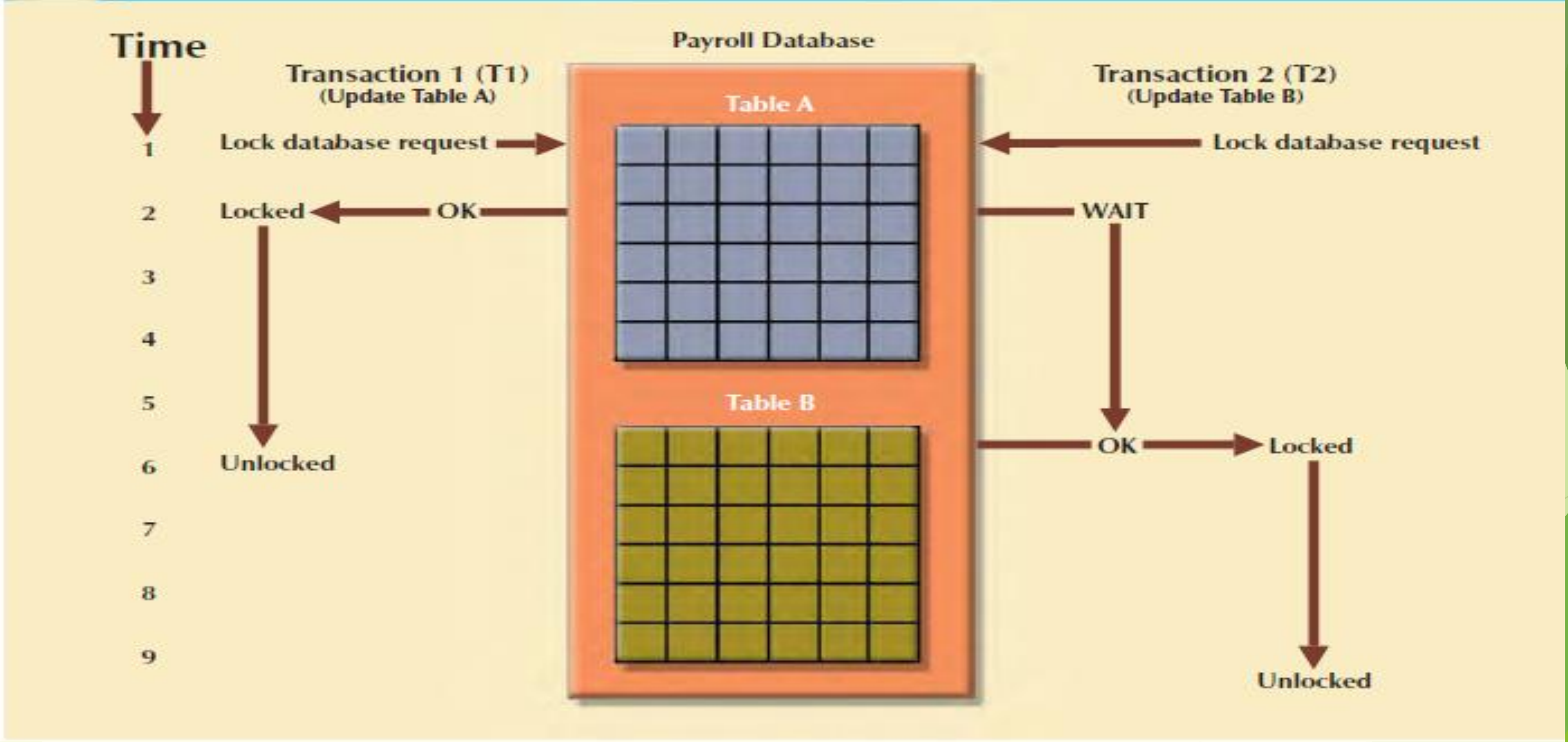
- Lock granularity indicates the level of lock use. Locking can take place at the following levels: database, table, page, row, or even field (attribute).

Database Level

- In a database-level lock, the entire database is locked, thus preventing the use of any tables in the database by transaction T2 while transaction Tl is being executed.
- This level of locking is good for batch processes, but it is unsuitable for multiuser DBMSs
- Because of the database-level lock, transactions T1 and T2 cannot access the same database concurrently even when they use different tables.

CONCURRENCY CONTROL

FIGURE 10.3 Database-level locking sequence



CONCURRENCY CONTROL

Table Level

- In a table-level lock, the entire table is locked, preventing access to any row by transaction T2 while transaction T1 is using the table.
- If a transaction requires access to several tables, each table may be locked. However, two transactions can access the same database as long as they access different tables.
- Table-level locks, while less restrictive than database-level locks, cause traffic jams when many transactions are waiting to access the same table
- Transactions T1 and T2 cannot access the same table even when they try to use different rows; T2 must wait until T1 unlocks the table

CONCURRENCY CONTROL

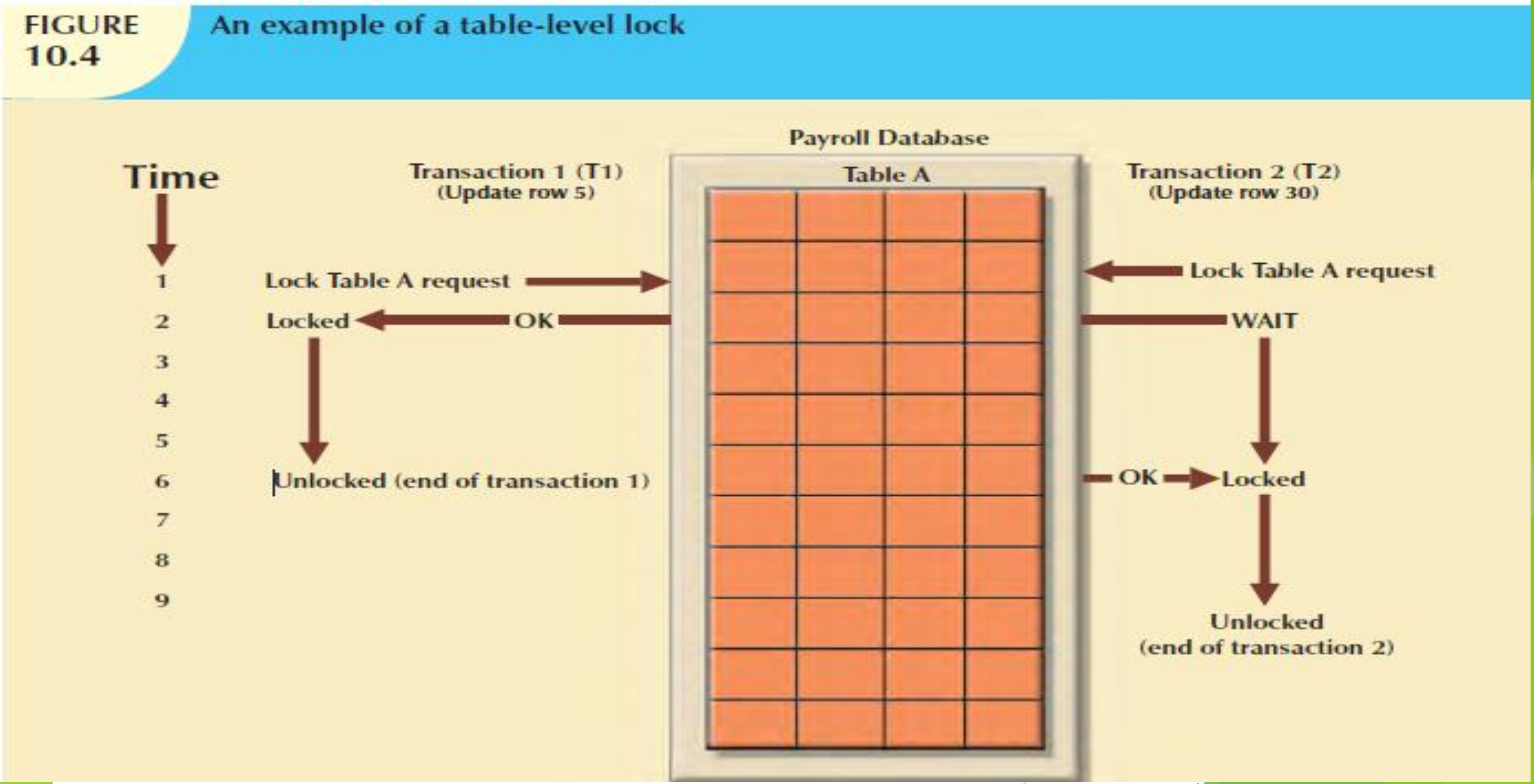


FIGURE 10.4 An example of a table-level lock

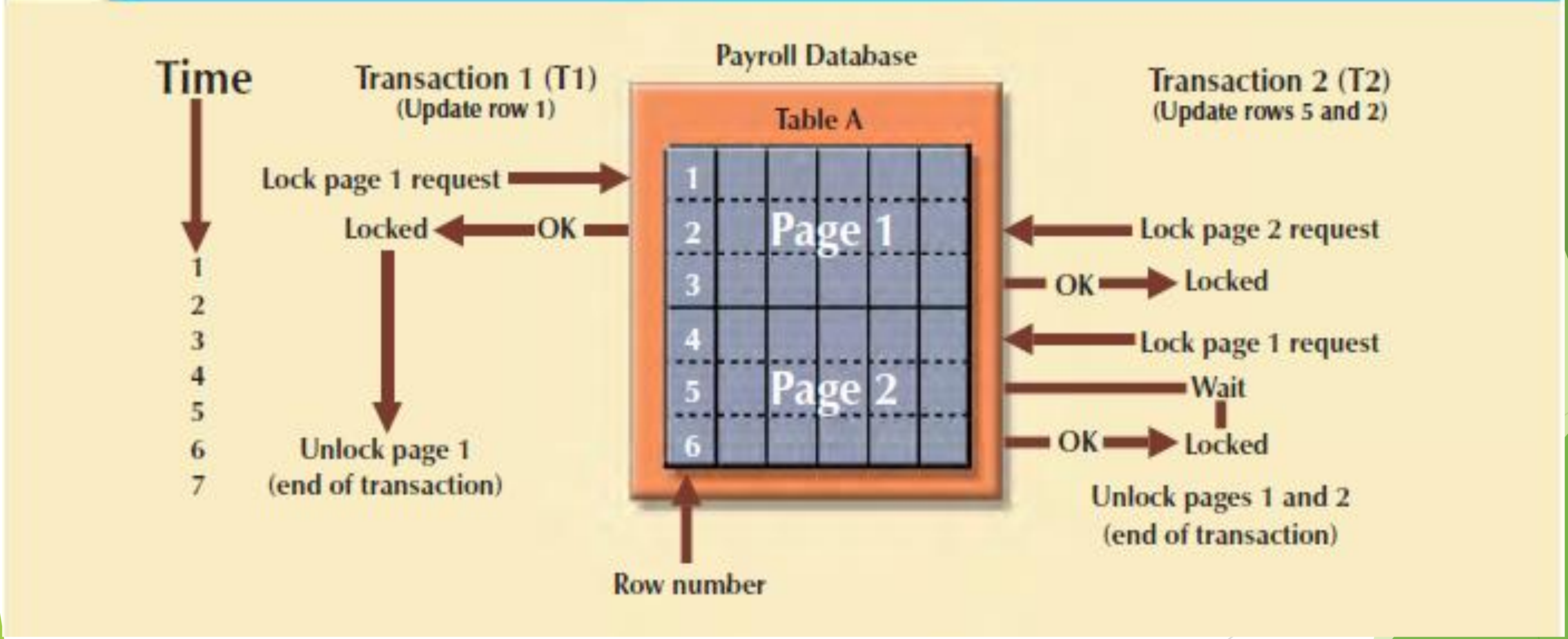
CONCURRENCY CONTROL

Page Level

- In a page-level lock, the DBMS will lock an entire diskpage.
- A diskpage, or page, is the equivalent of a *disk block*, which can be described as a directly addressable section of a disk.
- A page has a fixed size, such as 4K, 8K, or 16K.For example, if you want to write only 73 bytes to a 4K page, the entire 4K page must be read from disk, updated in memory, and written back to disk.
- A table can span several pages, and a page can contain several rows of one or more tables.
- Page-level locks are currently the most frequently used multiuser DBMS locking method

CONCURRENCY CONTROL

FIGURE 10.5 An example of a page-level lock



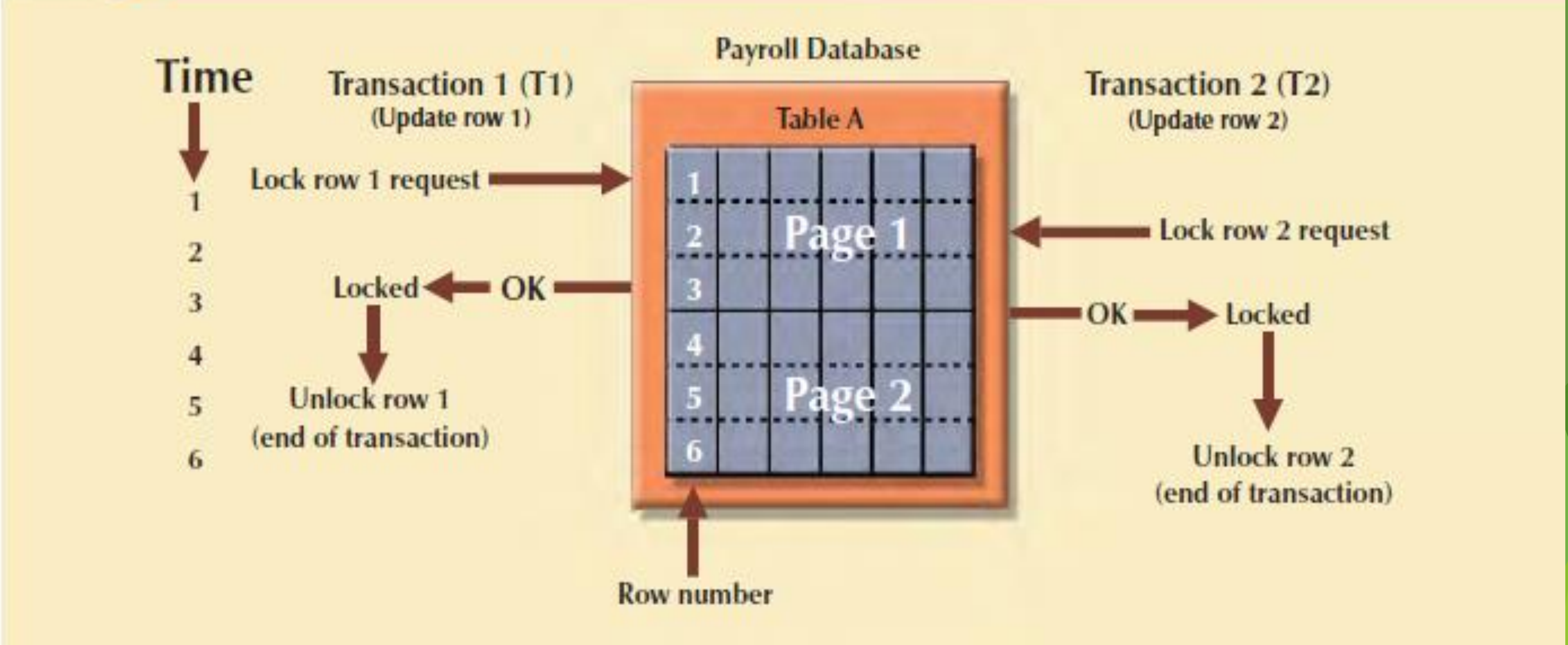
CONCURRENCY CONTROL

Row Level

- A **row-level lock is much less restrictive than the locks discussed earlier. The DBMS allows concurrent transactions** to access different rows of the same table even when the rows are located on the same page.
- Although the row-level locking approach improves the availability of data, its management requires high overhead because a lock exists for each row in a table of the database involved in a conflicting transaction
- Both transactions can execute concurrently, even when the requested rows are on the same page. T2 must wait only if it requests the same row as T1.

CONCURRENCY CONTROL

FIGURE 10.6 An example of a row-level lock



CONCURRENCY CONTROL

Field Level

- The **field-level lock allows concurrent transactions to access the same row as long as they require the use of different** fields (attributes) within that row.
- Although field-level locking clearly yields the most flexible multiuser data access, it is rarely implemented in a DBMS because it requires an extremely high level of computer overhead and because the row-level lock is much more useful in practice

CONCURRENCY CONTROL

Lock Types

Binary Locks

- A **binary lock has only two states: locked (1) or unlocked (0). If an object—that is, a database, table, page, or row—is** locked by a transaction, no other transaction can use that object.
- If an object is unlocked, any transaction can lock the object for its use. Every database operation requires that the affected object be locked.
- As a rule, a transaction must unlock the object after its termination.
- Therefore, every transaction requires a lock and unlock operation for each data item that is accessed. Such operations are automatically managed and scheduled by the DBMS; the user does not need to be concerned about locking or unlocking data items.

CONCURRENCY CONTROL

Lock Types

Binary Locks

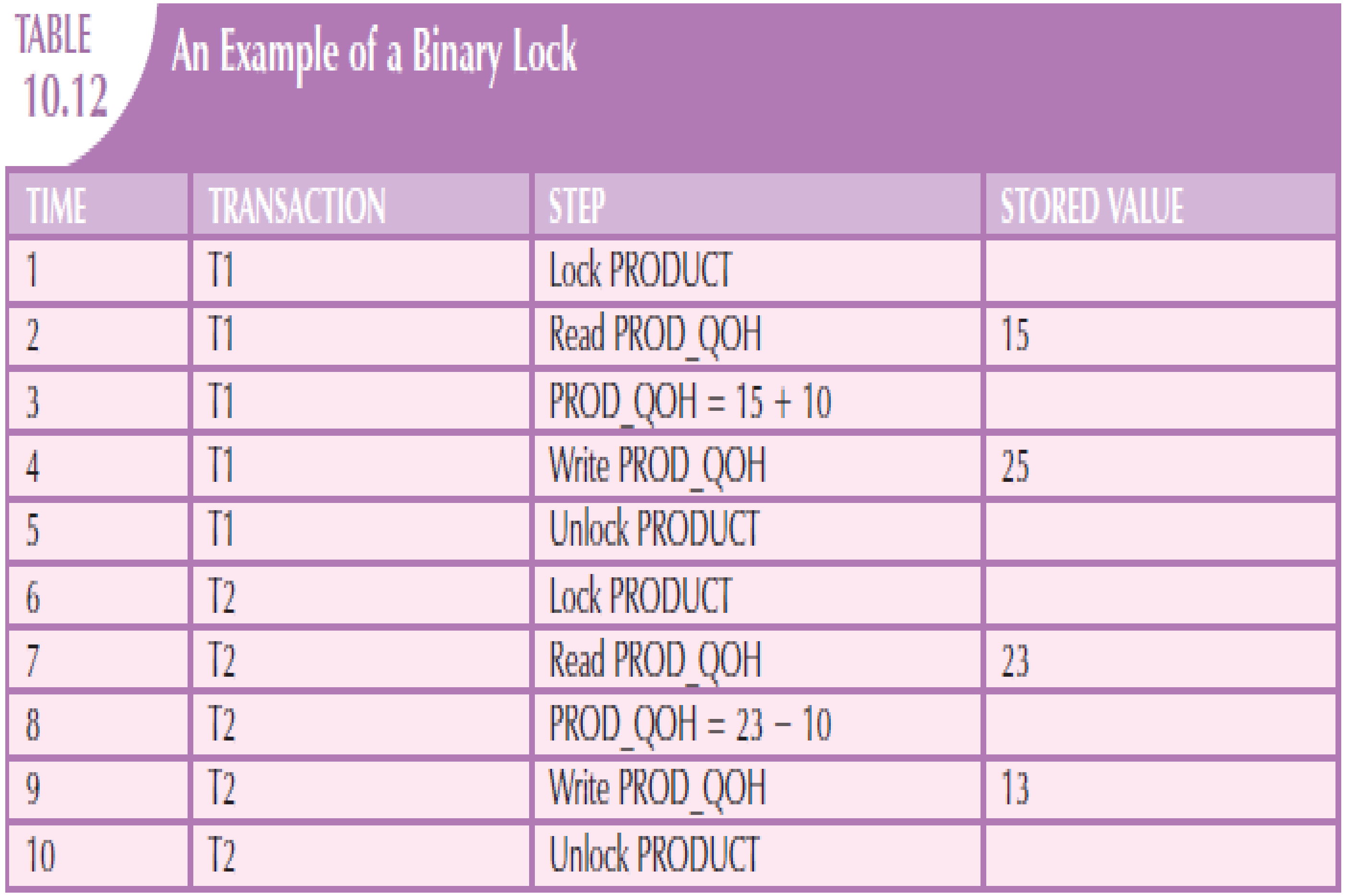


TABLE 10.12 An Example of a Binary Lock

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Lock PRODUCT	
2	T1	Read PROD_QOH	15
3	T1	PROD_QOH = 15 + 10	
4	T1	Write PROD_QOH	25
5	T1	Unlock PRODUCT	
6	T2	Lock PRODUCT	
7	T2	Read PROD_QOH	23
8	T2	PROD_QOH = 23 − 10	
9	T2	Write PROD_QOH	13
10	T2	Unlock PRODUCT	

CONCURRENCY CONTROL

Shared/Exclusive Locks

- The labels "shared" and "exclusive" indicate the nature of the lock. An **exclusive lock exists when access is reserved** specifically for the transaction that locked the object.
- The exclusive lock must be used when the potential for conflict exists.
- A **shared lock exists when concurrent transactions are granted read access** on the basis of a common lock.
- A shared lock produces no conflict as long as all the concurrent transactions are read-only
- A shared lock is issued when a transaction wants to read data from the database and no exclusive lock is held on that data item.
- An exclusive lock is issued when a transaction wants to update (write) a data item and no locks are currently held on that data item by any other transaction
- Using the shared/exclusive locking concept, a lock can have three states: unlocked, shared (read), and exclusive (write).

CONCURRENCY CONTROL

Shared/Exclusive Locks

- Two *transactions* conflict only when at least one of them is a Write transaction.
- Because the two Read transactions can be safely executed at once, shared locks allow several Read transactions to read the same data item concurrently.
- For example, if transaction T1 has a shared lock on data item X and transaction T2 wants to read data item X, T2 may also obtain a shared lock on data item X.
- If transaction T2 updates data item X, an exclusive lock is required by T2 over data item X.
- The exclusive lock is granted if and only if no other locks are held on the data item.
- Therefore, if a shared or exclusive lock is already held on data item X by transaction T1, an exclusive lock cannot be granted to transaction T2 and T2 must wait to begin until T1 commits
- *This condition* is known as the **mutual exclusive rule: only one transaction at a time can own an** exclusive lock on the same object.

CONCURRENCY CONTROL

Shared/Exclusive Locks

A shared/exclusive lock schema increases the lock manager’s overhead, for several reasons:

- The type of lock held must be known before a lock can be granted.
- Three lock operations exist: READ_LOCK (to check the type of lock), WRITE_LOCK (to issue the lock), and UNLOCK (to release the lock).
- The schema has been enhanced to allow a lock upgrade (from shared to exclusive) and a lock downgrade (from exclusive to shared)
- Although locks prevent serious data inconsistencies, they can lead to two major problems:
- The resulting transaction schedule might not be serializable.
- The schedule might create deadlocks. A **deadlock occurs when two transactions wait indefinitely for each** other to unlock data. A database **deadlock, which is equivalent to traffic gridlock in a big city, is caused when** two or more transactions wait for each other to unlock data.

CONCURRENCY CONTROL

Two-Phase Locking

Two-phase locking defines how transactions acquire and relinquish locks. Two-phase locking guarantees serializability, but it does not prevent deadlocks.

The two phases are:

1. A growing phase, in which a transaction acquires all required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.

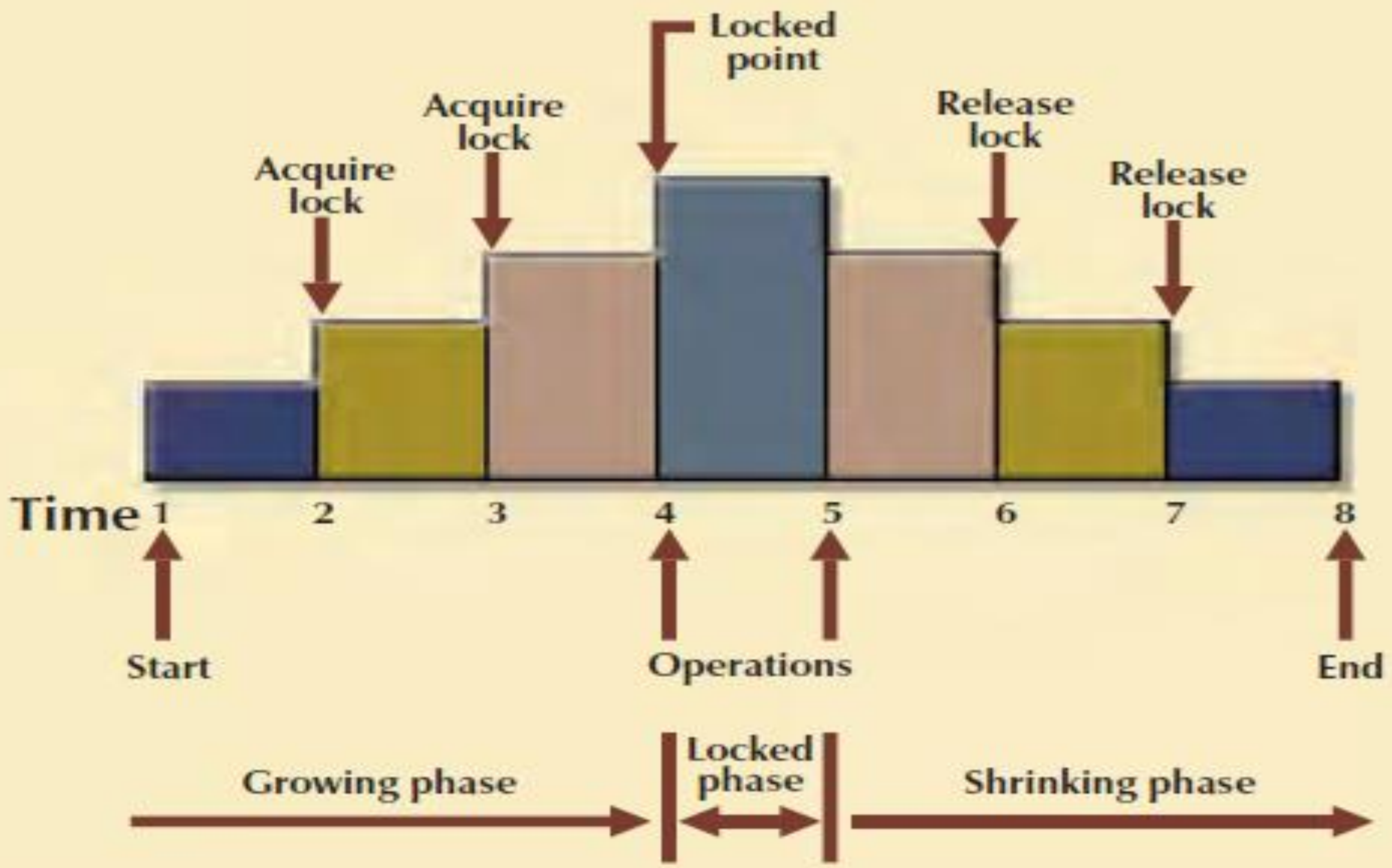
2. A shrinking phase, in which a transaction releases all locks and cannot obtain any new lock.

The two-phase locking protocol is governed by the following rules:

- Two transactions cannot have conflicting locks.
- No unlock operation can precede a lock operation in the same transaction.
- No data are affected until all locks are obtained—that is, until the transaction is in its locked point.

CONCURRENCY CONTROL

FIGURE 10.7 Two-phase locking protocol



CONCURRENCY CONTROL

Two-Phase Locking

- In this example, the transaction acquires all of the locks it needs until it reaches its locked point. (In this example, the transaction requires two locks.)
- When the locked point is reached, the data are modified to conform to the transaction's requirements. Finally, the transaction is completed as it releases all of the locks it acquired in the first phase

Deadlocks

- A deadlock occurs when two transactions wait indefinitely for each other to unlock data. For example, a deadlock occurs when two transactions, T1 and T2, exist in the following mode:
- T1 = access data items X and Y
- T2 = access data items Y and X
- If T1 has not unlocked data item Y, T2 cannot begin; if T2 has not unlocked data item X, T1 cannot continue.
- Consequently, T1 and T2 each wait for the other to unlock the required data item. Such a deadlock is also known as a **deadly embrace**.

Deadlocks

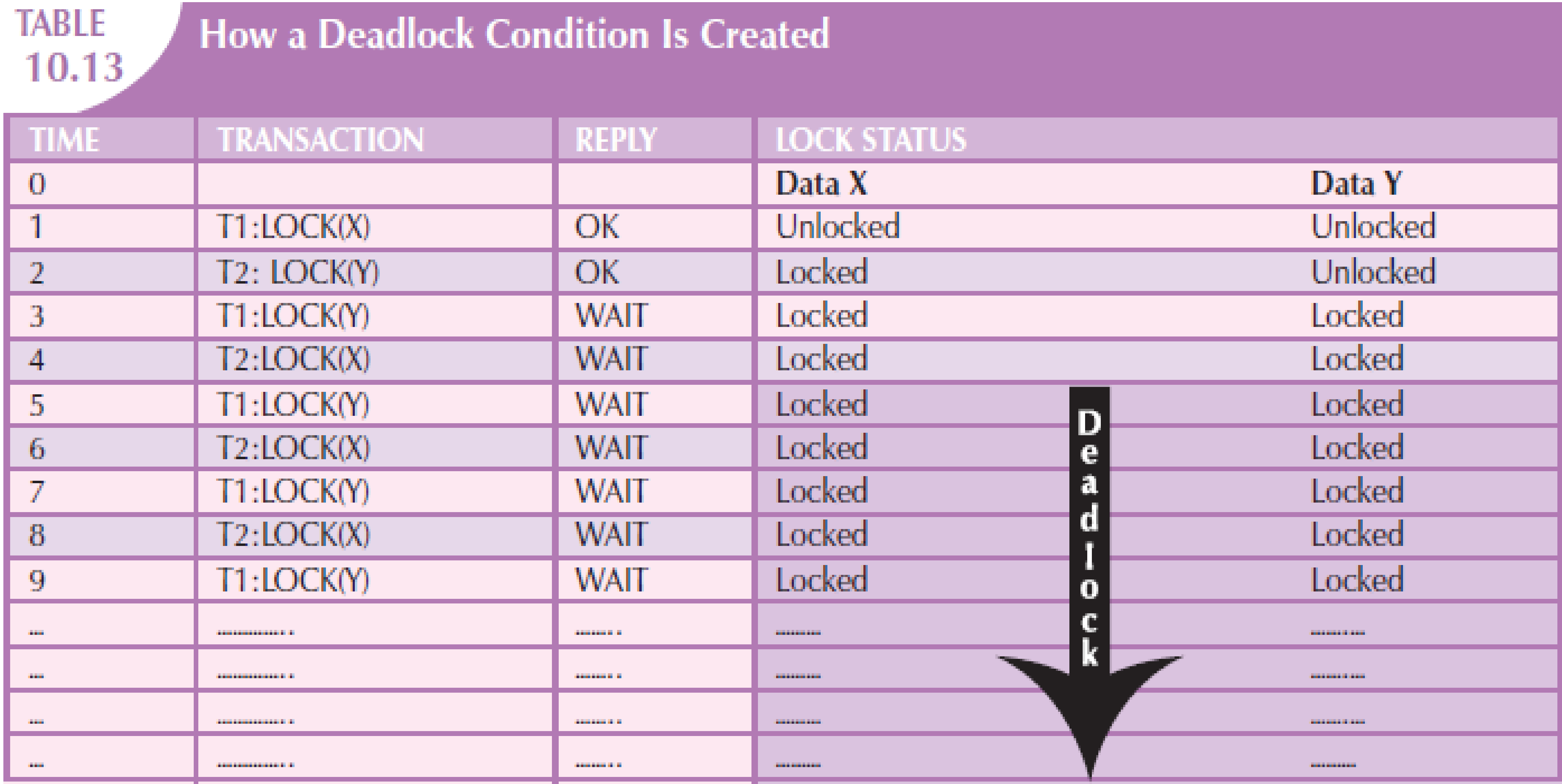


TABLE 10.13 How a Deadlock Condition Is Created

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
...	....	....	....	....
...	....	....	....	....
...	....	....	....	....
...	....	....	....	....

Deadlocks

The three basic techniques to control deadlocks are:

- *Deadlock prevention*. A transaction requesting a new lock is aborted when there is the possibility that a deadlock can occur. If the transaction is aborted, all changes made by this transaction are rolled back and all locks obtained by the transaction are released.
- The transaction is then rescheduled for execution. Deadlock prevention works because it avoids the conditions that lead to deadlocking.
- *Deadlock detection*. The DBMS periodically tests the database for deadlocks. If a deadlock is found, one of the transactions (the “victim”) is aborted (rolled back and restarted) and the other transaction continues.
- *Deadlock avoidance*. The transaction must obtain all of the locks it needs before it can be executed.
- This technique avoids the rolling back of conflicting transactions by requiring that locks be obtained in succession.
- However, the serial lock assignment required in deadlock avoidance increases action response times.

Deadlocks

CONCURRENCY CONTROL WITH TIME STAMPING METHODS

- The **time stamping** approach to scheduling concurrent transactions assigns a global, unique time stamp to each transaction.
- The time stamp value produces an explicit order in which transactions are submitted to the DBMS.
- Timestamps must have two properties: uniqueness and monotonicity. **Uniqueness** ensures that no equal time stamp values can exist, and **monotonicity**1 ensures that time stamp values always increase.
- All database operations (Read and Write) within the same transaction must have the same time stamp.
- The DBMS executes conflicting operations in time stamp order, thereby ensuring serializability of the transactions.
- If two transactions conflict, one is stopped, rolled back, rescheduled, and assigned a new time stamp value.

Deadlocks

CONCURRENCY CONTROL WITH TIME STAMPING METHODS

- The disadvantage of the time stamping approach is that each value stored in the database requires two additional timestamp fields: one for the last time the field was read and one for the last update.
- Time stamping thus increases memory needs and the database's processing overhead. Time stamping demands a lot of system resources because many transactions might have to be stopped, rescheduled, and restamped

Deadlocks

Wait/Die and Wound/Wait Schemes

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 short, in the **wait/die** scheme, the older transaction waits for the younger to complete and release its locks.

In the wound/wait scheme:

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

Deadlocks

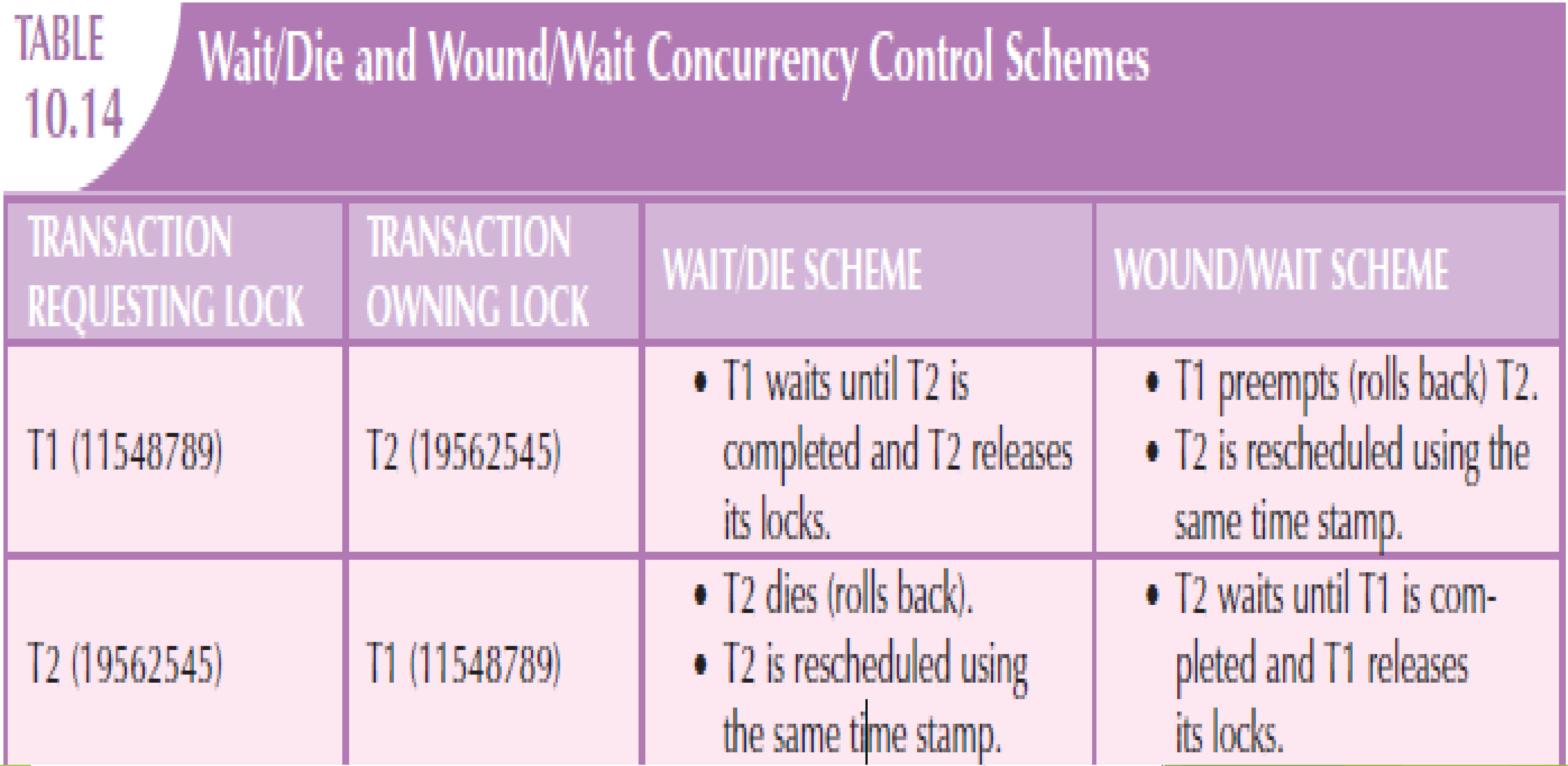


TABLE 10.14 Wait/Die and Wound/Wait Concurrency Control Schemes

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.

CONCURRENCY CONTROL WITH OPTIMISTIC METHODS

- The **optimistic approach** is based on the assumption that the majority of the database operations do not conflict.
- The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, 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 private copy of the database values.
- All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions

CONCURRENCY CONTROL WITH OPTIMISTIC METHODS

- 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.
- The optimistic approach is acceptable for most read or query database systems that require few update transactions.
- In a heavily used DBMS environment, the management of deadlocks—their prevention and detection—constitutes an important DBMS function

DATABASE RECOVERY MANAGEMENT

- **Database recovery** restores a database from a given state (usually inconsistent) to a previously consistent state.
- Recovery techniques are based on the **atomic transaction property**: all portions of the transaction must be treated as a single, logical unit of work in which all operations are applied and completed to produce a consistent database.
- If, for some reason, any transaction operation cannot be completed, the transaction must be aborted and any changes to the database must be rolled back (undone). In short, transaction recovery reverses all of the changes that the transaction made to the database before the transaction was aborted.
- Recovery techniques also apply to the *database* and to the *system* after some type of critical error has occurred.
- Critical events can cause a database to become nonoperational and compromise the integrity of the data

DATABASE RECOVERY MANAGEMENT

- Examples of critical events are:
- *Hardware/software failures*. A failure of this type could be a hard disk media failure, a bad capacitor on a motherboard, or a failing memory bank.
- Other causes of errors under this category include application program or operating system errors that cause data to be overwritten, deleted, or lost.
- Some database administrators argue that this is one of the most common sources of database problems.
- *Human-caused incidents*. This type of event can be categorized as unintentional or intentional.
- - An unintentional failure is caused by carelessness by end users. Such errors include deleting the wrong rows from a table, pressing the wrong key on the keyboard, or shutting down the main database server by accident.

DATABASE RECOVERY MANAGEMENT

- Examples of critical events are:
- Intentional events are of a more severe nature and normally indicate that the company data are at serious risk.
- Under this category are security threats caused by hackers trying to gain unauthorized access to data resources and virus attacks caused by disgruntled employees trying to compromise the database operation and damage the company.
- *Natural disasters*. This category includes fires, earthquakes, floods, and power failures.

Transaction Recovery

- Database transaction recovery uses data in the transaction log to recover a database from an inconsistent state to a consistent state.
- Four important concepts that affect the recovery process:
- The **write-ahead-log protocol** ensures that transaction logs are always written *before* any database data are actually updated. This protocol ensures that, in case of a failure, the database can later be recovered to a consistent state, using the data in the transaction log.
- **Redundant transaction logs** (several copies of the transaction log) ensure that a physical disk failure will not impair the DBMS’s ability to recover data
- Database **buffers** are temporary storage areas in primary memory used to speed up disk operations.
- To improve processing time, the DBMS software reads the data from the physical disk and stores a copy of it on a “buffer” in primary memory

Transaction Recovery

- The database recovery process involves bringing the database to a consistent state after a failure.
- Transaction recovery procedures generally make use of deferred-write and write-through techniques.
- When the recovery procedure uses a **deferred-write technique** (also called a **deferred update**), the transaction operations do not immediately update the physical database. Instead, only the transaction log is updated.
- The database is physically updated only after the transaction reaches its commit point, using information from the transaction log.
- If the transaction aborts before it reaches its commit point, no changes (no ROLLBACK or undo) need to be made to the database because the database was never updated.
- The recovery process for all started and committed transactions (before the failure) follows these steps:

Transaction Recovery

- When a transaction updates data, it actually updates the copy of the data in the buffer because that process is much faster than accessing the physical disk every time.
- Later on, all buffers that contain updated data are written to a physical disk during a single operation, thereby saving significant processing time
- Database **checkpoints** are operations in which the DBMS writes all of its updated buffers to disk. While this is happening, the DBMS does not execute any other requests.
- A checkpoint operation is also registered in the transaction log. As a result of this operation, the physical database and the transaction log will be in sync.
- This synchronization is required because update operations update the copy of the data in the buffers and not in the physical database.
- Checkpoints are automatically scheduled by the DBMS several times per hour.

Transaction Recovery

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that performed a commit operation after the last checkpoint, the DBMS uses the transaction log records to redo the transaction and to update the database, using the “after” values in the transaction log. The changes are made in ascending order, from oldest to newest.

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, nothing needs to be done because the database was never updated.

Transaction Recovery

- When the recovery procedure uses a **write-through technique** (also called an **immediate update**), the database is immediately updated by transaction operations during the transaction's execution, 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 transaction log "before" values
- The recovery process follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data were physically savedto disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data are already saved.

3. For a transaction that was committed after the last checkpoint, the DBMS redoes the transaction, using the "after" values of the transaction log. Changes are applied in ascending order, from oldest to newest.

Transaction Recovery

4. For any transaction that had a ROLLBACK operation after the last checkpoint or that was left active (with neither a COMMIT nor a ROLLBACK) before the failure occurred, the DBMS uses the transaction log records to ROLLBACK or undo the operations, using the “before” values in the transaction log. Changes are applied in reverse order, from newest to oldest