ModuleIII

TRANSACTION MANAGEMENT & CONCURRENCY CONTROL

- **Module III:**
- **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

Transactio n

- Collections of operations that form a single logical unit of work are called **transactions**.
- A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does.
- **Transaction** is a unit of program execution that accesses and possibly updates various data items.

What is a Transaction?

- Any action that reads from and/or writes to a database may consist of
  - Simple SELECT statement to generate a list of table contents
  - A series of related UPDATE statements to change the values of attributes in various tables
  - A series of INSERT statements to add rows to one or more tables
  - A combination of SELECT, UPDATE, and INSERT statements

What is a Transaction?

- A *logical* unit of work that must be either entirely completed or aborted
- Successful transaction changes the database from one *consistent* state to another
  - One in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests
  - The equivalent of a single SQL statement in an application program or transaction

Evaluating Transaction Results

- Not all transactions update the database
- SQL code represents a transaction because database is accessed
- Improper or incomplete transactions can have a devastating effect on database integrity
  - Some DBMSs provide means by which user can define enforceable constraints based on business rules
  - Other integrity rules are enforced automatically by the DBMS when table structures are properly defined, thereby letting the DBMS validate some transactions

Transaction Properties(ACID properties)

- **Atomicity**
  - Requires that ***all*** operations (SQL requests) of a transaction be completed; if not, then the transaction is aborted
  - A transaction is treated as a single, indivisible, logical unit of work
  - This “**all-or-none**” property is referred to as atomicity.

Consistency

- Consistency property ensures that the database must remain in **the consistent state before the start of transaction and after the transaction is over.**
- Consistency states that only valid data will be written to the database.
- If for some reason a transaction is executed that violates the database consistency rules the entire transaction will be rolled back.

- **Isolation**
  - Data used during execution of a transaction cannot be used by second transaction until first one is completed
  - **Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions *Ti and Tj , it appears* to *Ti that either Tj finished execution before Ti started, or Tj started execution* after *Ti finished.***

Durability

- After a transaction completes successfully, **the changes it has made to the database persist**, even if there are **system failures**.
- Durability can be implemented by writing all transaction into a **transaction log** that can be used to roll back a system state right before failure.
- A transaction can only regard as committed after it is written safely in the log.
- For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

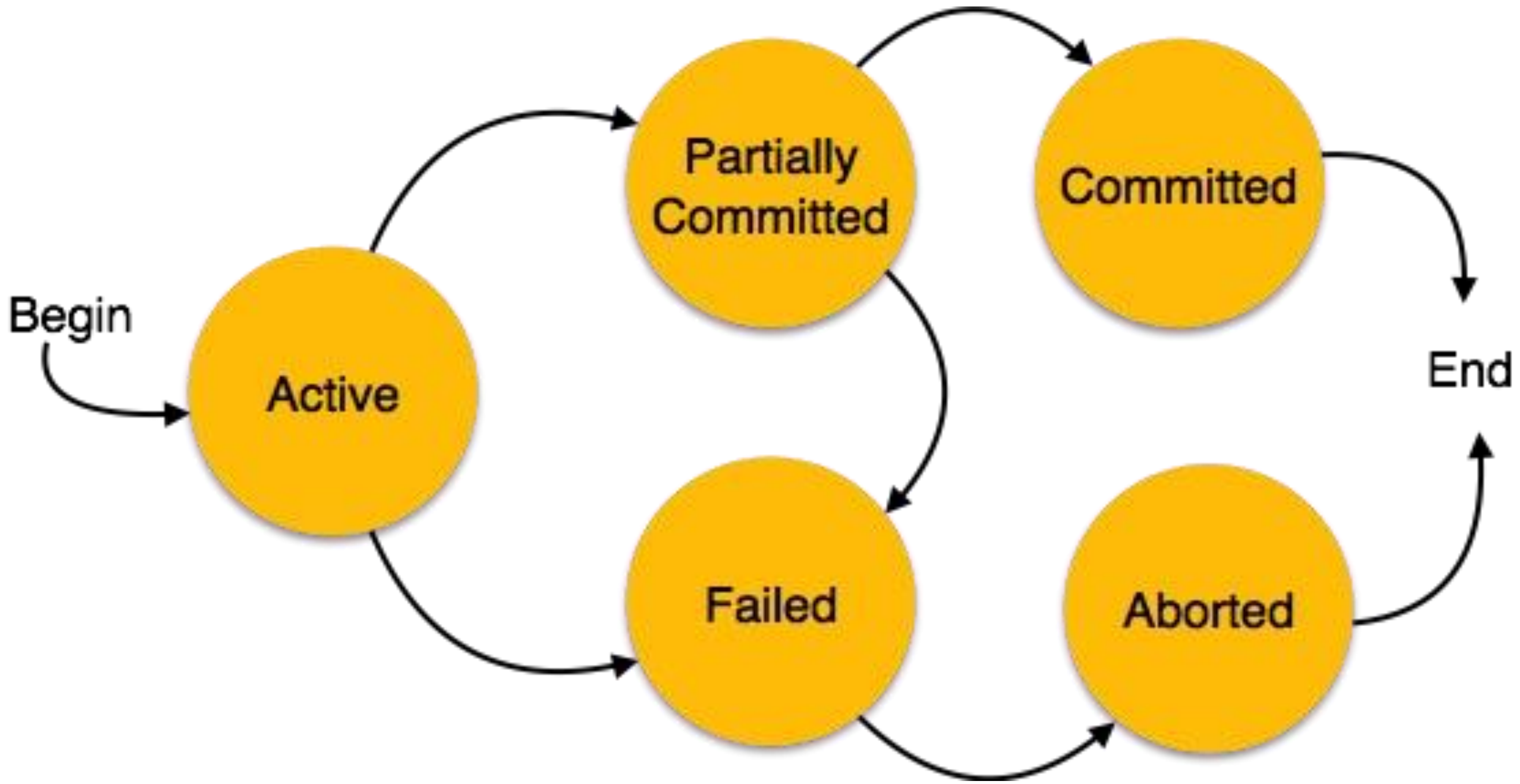
- These properties are called the **ACID properties.**

Transaction State

- A transaction must be in one of the following states:
  - **Active:-** The initial state; the transaction stays inthis state while it is executing.
- **Partially committed:-**
  - After the final statement has been executed.
- **Failed:-**
  - After the discovery that normal execution can no longer proceed.

Transaction State

- **Aborted:-**
  - After the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- **Committed:-**
  - After successful completion


Begin
Active
Partially Committed
Committed
Failed
Aborted
End

Transaction Management with SQL

- ANSI has defined standards that govern SQL database transactions
- Transaction support is provided by two SQL statements: COMMIT and ROLLBACK
- ANSI standards require that, when a transaction sequence is initiated by a user or an application program,it must continue through all succeeding SQL statements until one of four events occurs

Transaction Management with SQL

1. A COMMIT statement is reached- all changes are permanently recorded within the database
2. A ROLLBACK is reached – all changes are aborted and the database is restored to a previous consistent state
3. The end of the program is successfully reached – equivalent to a COMMIT
4. The program abnormally terminates and a rollback occurs

The Transaction Log

- Keeps track of all transactions that update the database. It contains:
  - A record for the beginning of transaction
  - For each transaction component (SQL statement)
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - “Before” and “after” values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction
- Increases processing overhead but the ability to restore a corrupted database is worth the price

The Transaction Log

- Increases processing overhead but the ability to restore a corrupted database is worth the price
- If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to a previous state
- The log it itself a database and to maintain its integrity many DBMSs will implement it on several different disks to reduce the risk of system failure

A Transaction Log

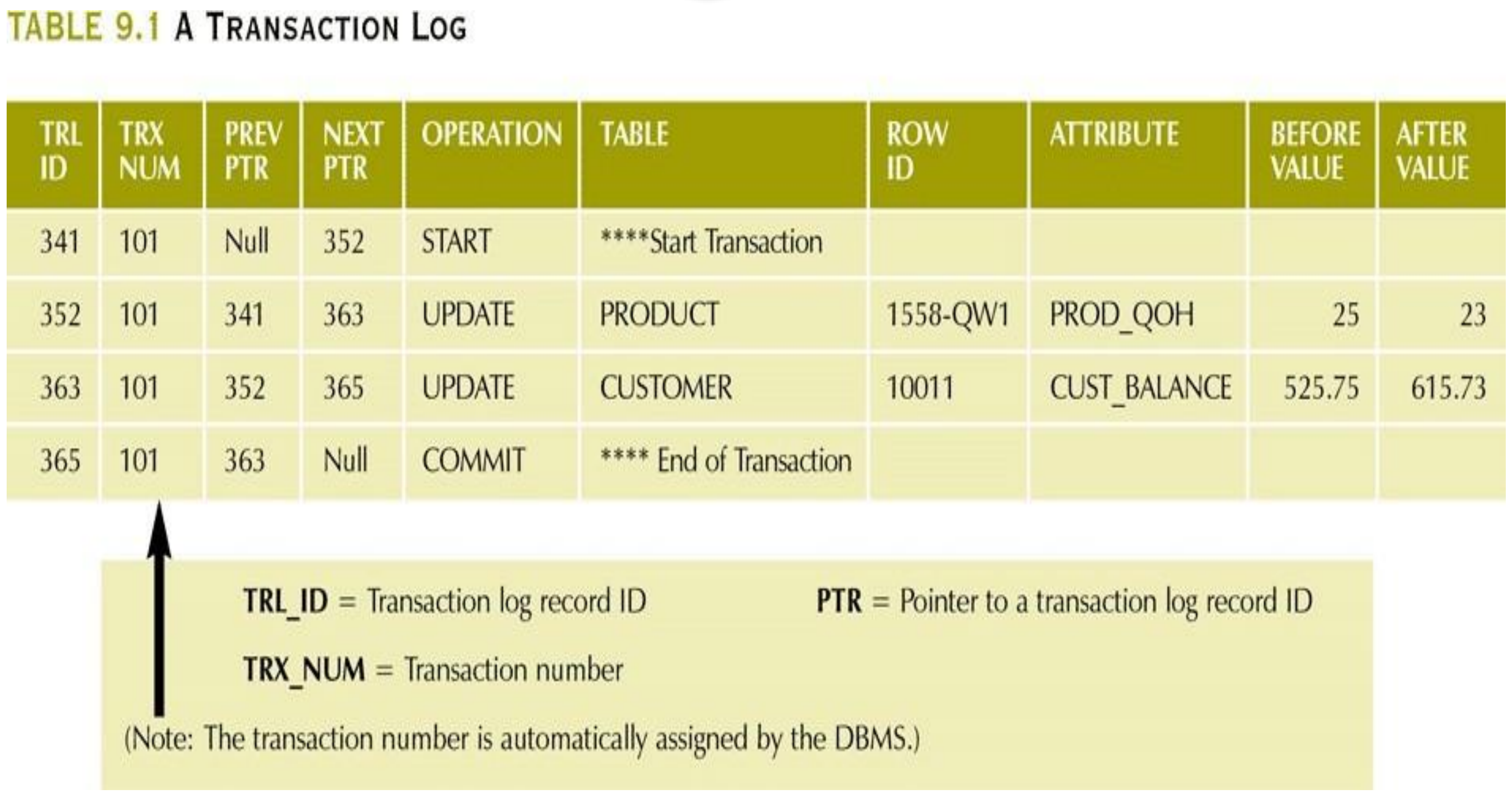


TABLE 9.1 A TRANSACTION LOG

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

PTR = Pointer to a transaction log record ID

TRX_NUM = Transaction number

(Note: The transaction number is automatically assigned by the DBMS.)

concurrency control

- Coordinating 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.
- To achieve this goal, most concurrency control techniques are oriented toward preserving the isolation property of concurrently executing transactions.
- 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, uncom mitted data, and inconsistent retrievals.

lost update

- 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).
- To see an illustration of lost updates, examine a simple PROD- UCT table. One of the table's attributes is a product's quantity on hand (PROD_ QOH). Assume that you have a product whose current PROD_QOH value is 35. Also assume that two concurrent transactions, T1 and T2, occur and update the PROD_QOH value for some item in the PRODUCT table. The transactions are shown in Table 10.2.
-

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

Table 10.3 shows the serial execution of the transactions under normal circumstances, yielding the correct answer PROD_QOH = 105.

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

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

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 iso- lation property of transactions.
- To illustrate that possibility, use the same transactions described during the lost updates discussion. T1 has two atomic parts, one of which is the update of the inventory; the other possible part is the update of the invoice total (not shown). T1 is forced to roll back due to an error during the updating of the invoice's total; it rolls back all the way, undoing the inventory update as well. This time the T1 transac- tion is rolled back to eliminate the addition of the 100 units. (See Table 10.5.) Because T2 subtracts 30 from the original 35 units, the correct answer should be 5.

Table 10.6 shows how the serial execution of these transactions yields the correct answer under normal circumstances.

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

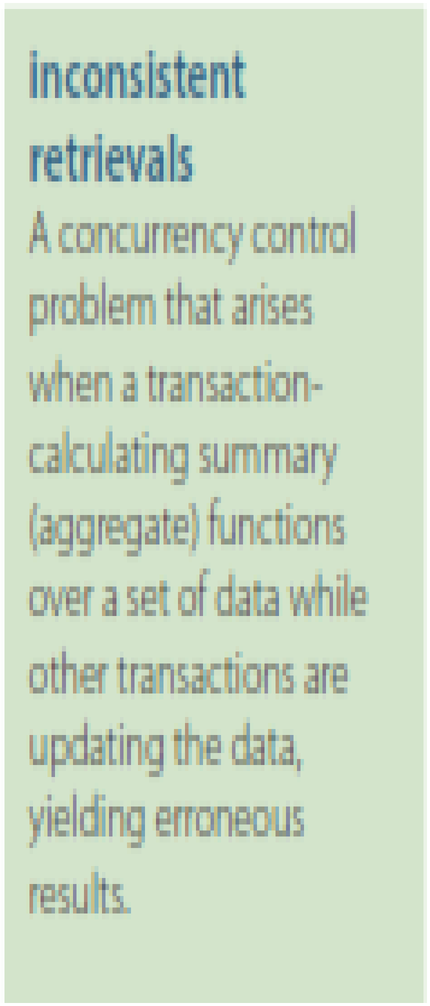
Table 10.7 shows how the uncommitted data problem can arise when the ROLLBACK is completed after T2 has begun its execution.

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

Inconsistent retrievals



inconsistent retrievals
A concurrency control problem that arises when a transaction-calculating summary (aggregate) functions over a set of data while other transactions are updating the data, yielding erroneous results.

- **Inconsistent retrievals** occur when a transaction accesses data before and after one or more other transactions finish working with such data.
- For example, an inconsistent retrieval would occur if transaction T1 calculated some summary (aggregate) function over a set of data while another transaction (T2) was updating the same data. The problem is that the transaction might read some data before it is changed and other data *after* it is changed, thereby yielding inconsistent results.
- To illustrate the problem, assume the following conditions:
- T1 calculates the total quantity on hand of the products stored in the PRODUCT table.
- At the same time, T2 updates the quantity on hand (PROD_QOH) for two of the PRODUCT table's products.
- The two transactions are shown in Table 10.8.

TABLE 10.8

RETRIEVAL DURING UPDATE

TRANSACTION T1	TRANSACTION T2

SELECT SUM(PROD_QOH) FROM PRODUCT	UPDATE PRODUCT
	SET PROD_QOH = PROD_QOH + 10
	WHERE PROD_CODE = 1546-QQ2

	UPDATE PRODUCT
	SET PROD_QOH = PROD_QOH – 10
	WHERE PROD_CODE = 1558-QW1

	COMMIT;

TABLE 10.9

TRANSACTION RESULTS: DATA ENTRY CORRECTION

	BEFORE	AFTER
PROD_CODE	PROD_QOH	PROD_QOH
11QER/31	8	8
13-Q2/P2	32	32
1546-QQ2	15	(15 + 10) $\longrightarrow$ 25
1558-QW1	23	(23 – 10) $\longrightarrow$ 13
2232-QTY	8	8
2232-QWE	6	6
Total	**92**	**92**

TABLE 10.10

INCONSISTENT RETRIEVALS

TIME	TRANSACTION	ACTION	VALUE	TOTAL
1	T1	Read PROD_QOH for PROD_CODE = '11QER/31'	8	8
2	T1	Read PROD_QOH for PROD_CODE = '13-Q2/P2'	32	40
3	T2	Read PROD_QOH for PROD_CODE = '1546-QQ2'	15	
4	T2	PROD_QOH = 15 + 10		
5	T2	Write PROD_QOH for PROD_CODE = '1546-QQ2'	25	
6	T1	Read PROD_QOH for PROD_CODE = '1546-QQ2'	25	(After) 65
7	T1	Read PROD_QOH for PROD_CODE = '1558-QW1'	23	(Before) 88
8	T2	Read PROD_QOH for PROD_CODE = '1558-QW1'	23	
9	T2	PROD_QOH = 23 – 10		
10	T2	Write PROD_QOH for PROD_CODE = '1558-QW1'	13	
11	T2	***** COMMIT *****		
12	T1	Read PROD_QOH for PROD_CODE = '2232-QTY'	8	96
13	T1	Read PROD_QOH for PROD_CODE = '2232-QWE'	6	102

Scheduler

- 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 a transaction updates multiple tables and rows. (If the transaction contains only one update, then there is no temporary inconsistency.)
- The temporary inconsistency exists because a computer executes the operations serially, one after another. During this serial process, the isolation property of transactions prevents them from accessing the data not yet released by other transactions.
- This consideration is even more important today, with the use of multicore processors that can execute several instructions at the same time.
- What would happen if two transactions executed concurrently and they were accessing the same data?

Need for a scheduler

- As long as two transactions, T1 and T2, access *unrelated* data, there is no conflict and the order of execution is irrelevant to the final outcome.
- However, if the transactions operate on related data or the same data, conflict is possible among the transaction components and the selection of one execution order over another might have some undesirable consequences. So, **how is the correct order determined, and who determines that order?**
- Fortunately, the DBMS handles that tricky assignment by using a built-in scheduler.

- **The scheduler is a special DBMS process that establishes the order in which the operations are executed within concurrent transactions.**
- 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 methods
- **not all transactions are serializable.** 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, 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).

schedulercontd

- also makes sure that the computer’s central processing unit (CPU) and storage systems are used efficiently.
- **If there were no way to schedule the execution of transactions, all of them would be executed on a first-come, first-served basis.**
  - The problem with that approach is that processing time is wasted when the CPU waits for a READ or WRITE operation to finish, thereby losing several CPU cycles.
  - le first-come, first-served scheduling tends to yield unacceptable response times within the multiuser DBMS environment. Therefore, some other scheduling method is needed to improve the efficiency of the overall system.

- Additionally, the scheduler facilitates data isolation to ensure that two transactions do not update the same data element at the same time.
  - Database operations might require READ and/or WRITE actions that produce conflicts.
  - For example, Table 10.11 shows the possible conflict scenarios when two transactions, T1 and T2, are executed concurrently over the same data. Note that in Table 10.11, two operations are in conflict when they access the same data and at least one of them is a WRITE operation.

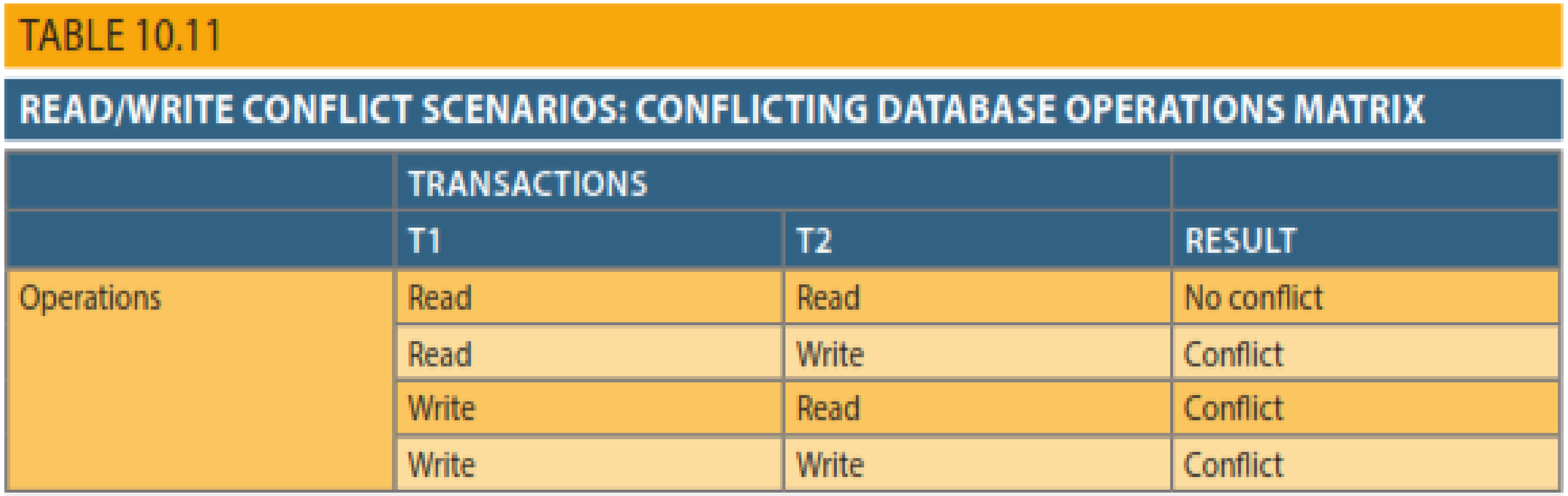


TABLE 10.11

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 using locks

- Locking methods facilitate the isolation of data items used in concurrently executing transactions.
- 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 concurrent transactions might attempt to manipulate the same data item at the same time.

- **The use of locks based on the assumption that conflict between transactions is likely is usually referred to as pessimistic locking.**
- Most multiuser DBMSs automatically initiate and enforce locking procedures. All lock information is handled by a **lock manager**, which is responsible for assigning and policing the locks used by the transactions.

- **The use of locks based on the assumption that conflict between transactions is likely is usually referred to as pessimistic locking.**
- Most multiuser DBMSs automatically initiate and enforce locking procedures. All lock information is handled by a **lock manager**, which is responsible for assigning and policing the locks used by the transactions.

Lock granularity

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

database-level lock

- the entire database is locked
- preventing the use of any tables in the database by transaction T2 while transaction T1 is being executed.
- good for batch processes
- it is unsuitable for multiuser DBMSs.
- slow data access would be if thousands of transactions had to wait for the previous transaction to be completed before the next one could reserve the entire database.
- Figure 10.3 illustrates the database-level lock; because of it, transactions T1 and T2 cannot access the same database concurrently *even when they use different tables.*

FIGURE 10.3 DATABASE-LEVEL LOCKING SEQUENCE

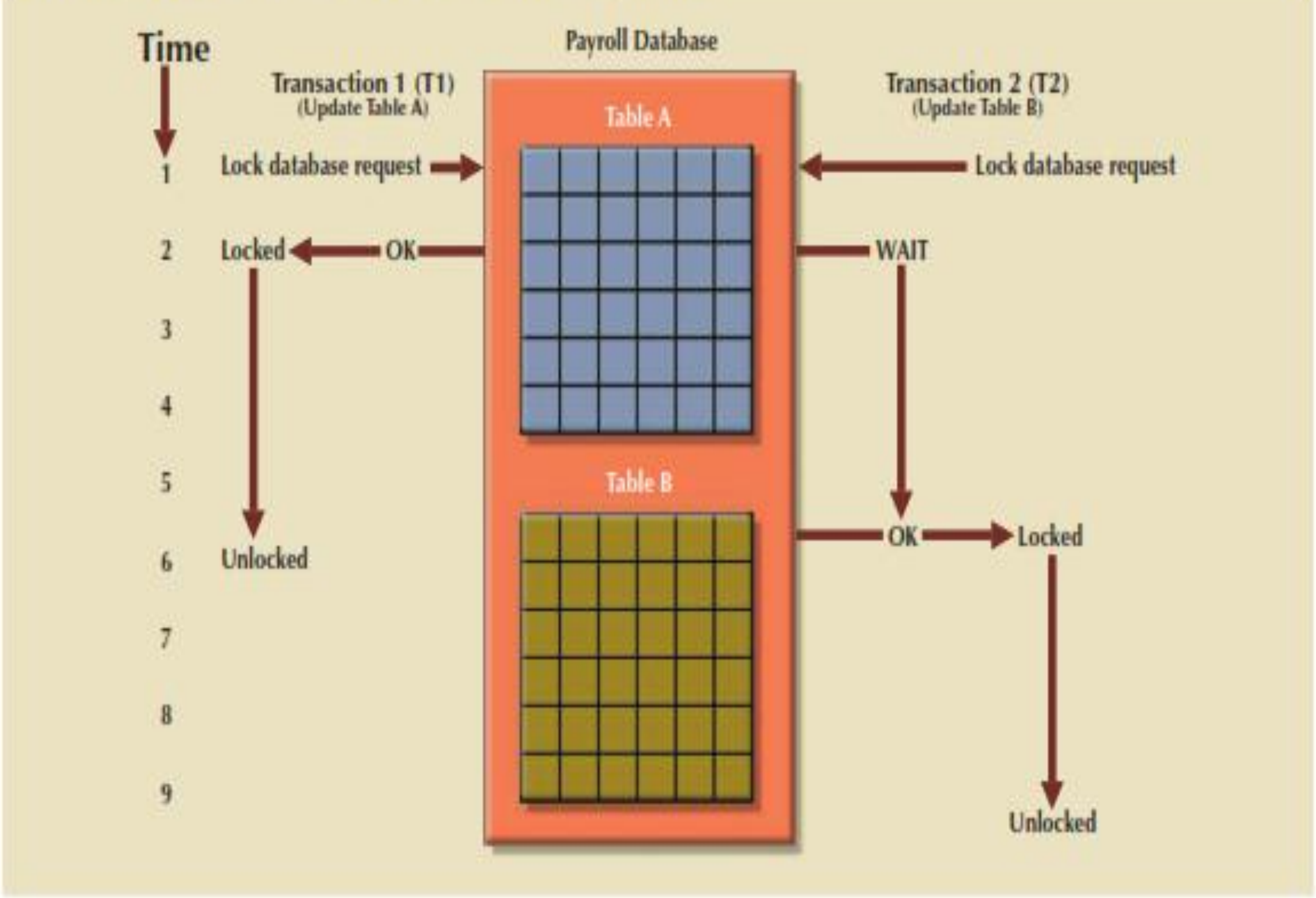
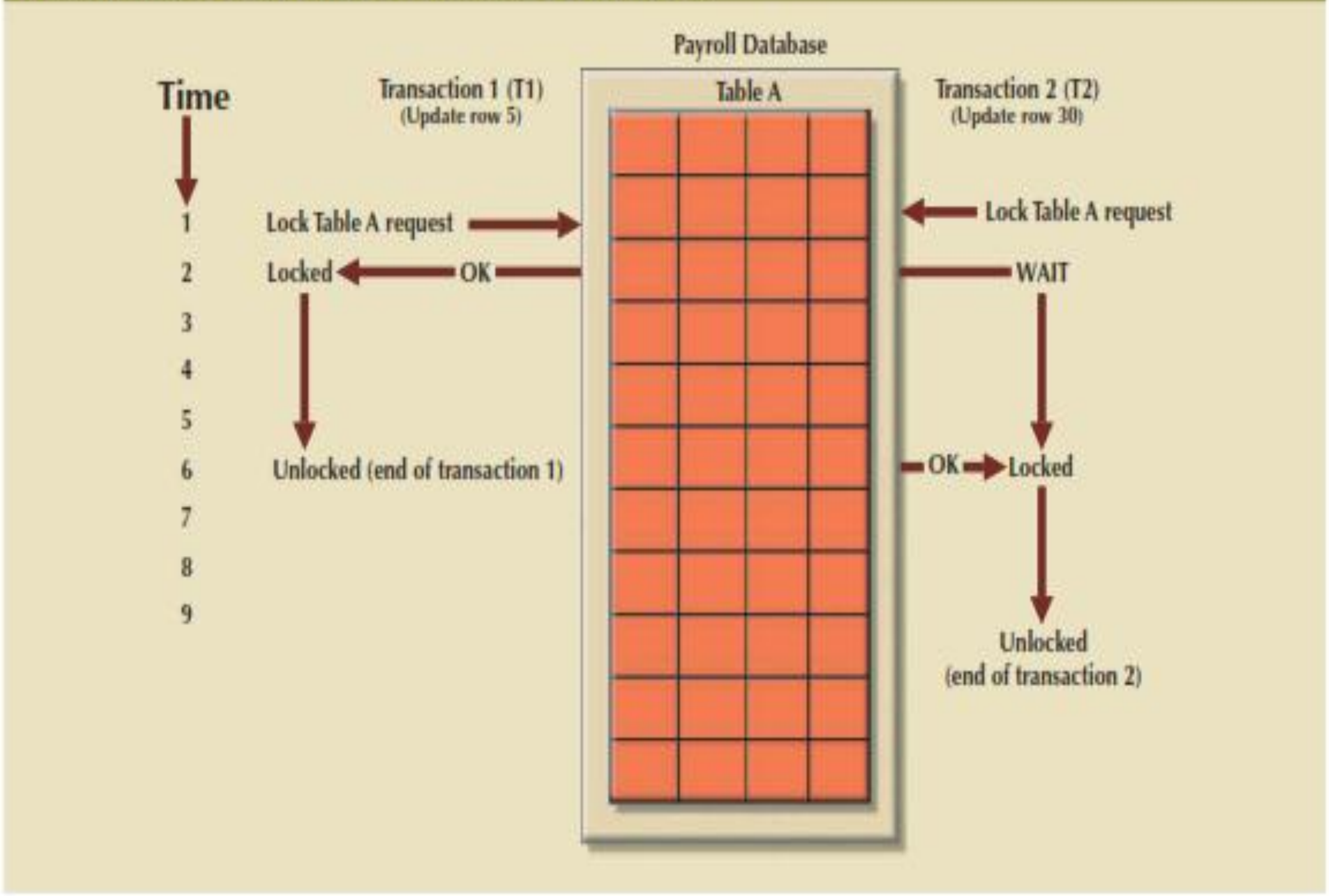


table-level lock

- 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. Such a condition is especially irksome if the lock forces a delay when different transactions **require access to different parts of the same table** that is, when the transactions would not interfere with each other. Consequently, table-level locks are not suitable for multiuser DBMSs.
- Figure 10.4 illustrates the effect of a table-level lock. Note that 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.

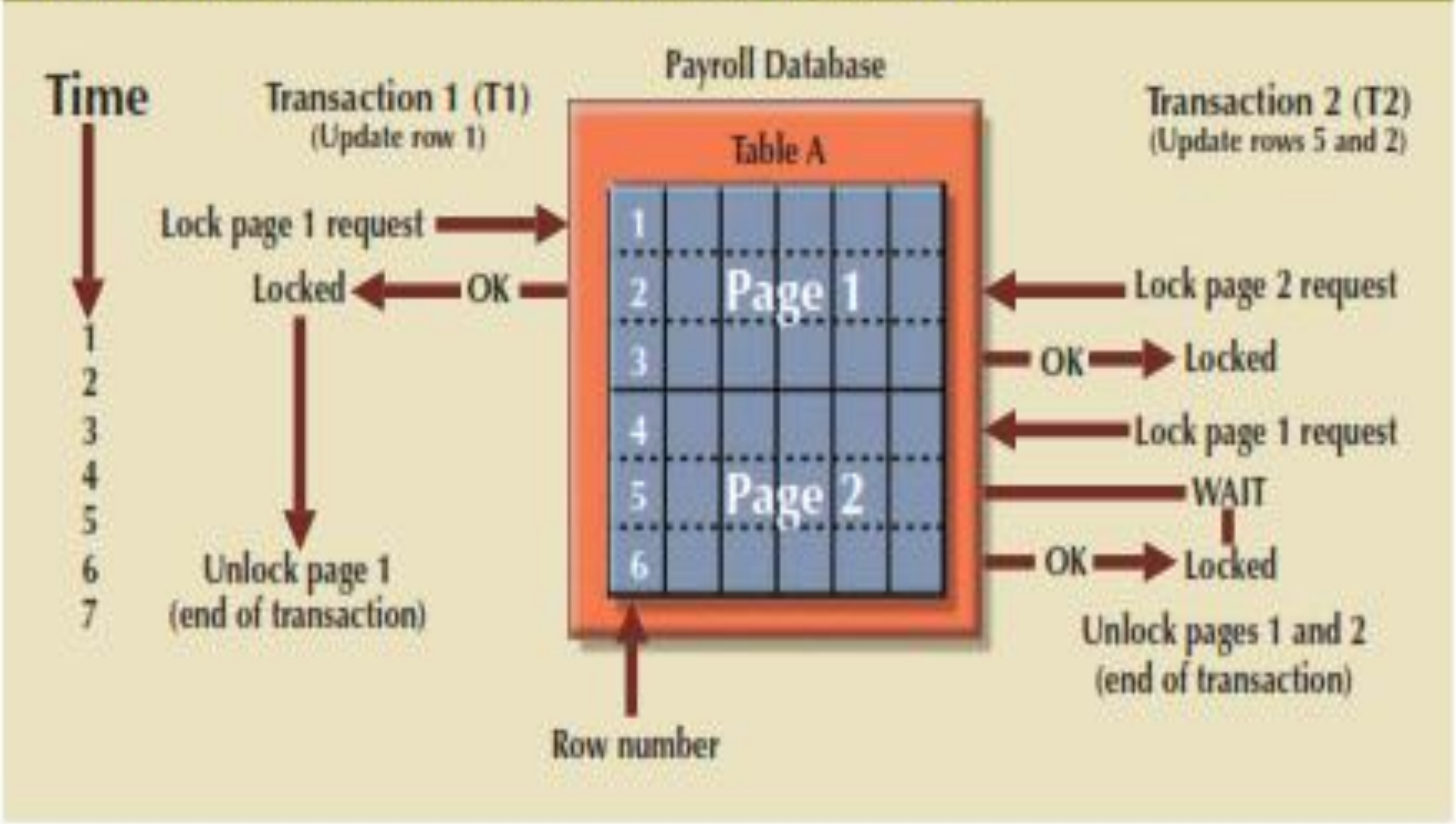
FIGURE 10.4 AN EXAMPLE OF A TABLE-LEVEL LOCK



page-level lock,

- locks an entire diskpage.
- A **diskpage**, or **page**, is the equivalent of a *diskblock*, 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 locking method for multiuser DBMSs.**
- An example of a page-level lock is shown in Figure 10.5. Note that T1 and T2 access the same table while locking different diskpages. If T2 requires the use of a row located on a page that is locked by T1, T2 must wait until T1 unlocks the page.

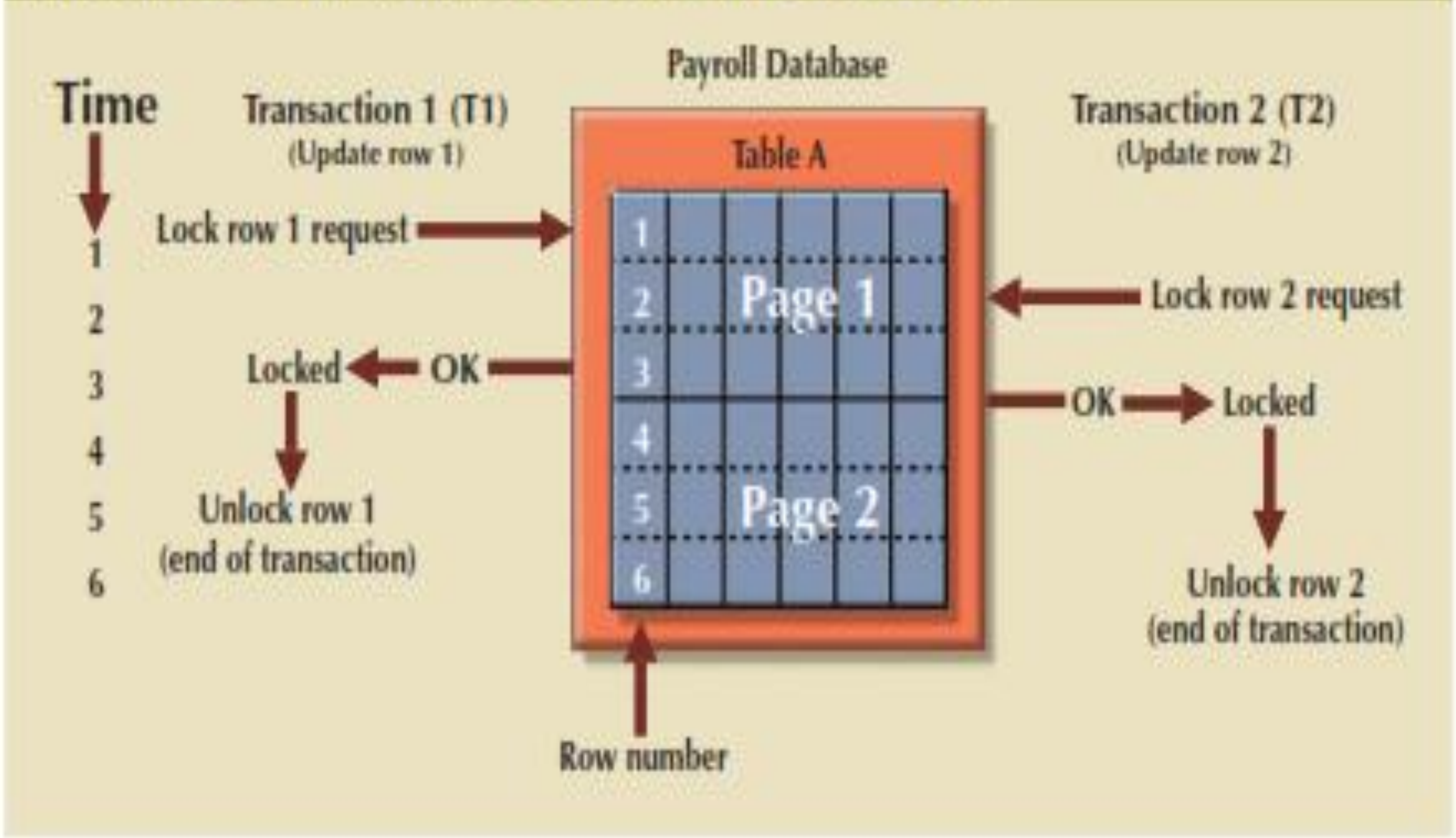
FIGURE 10.5 AN EXAMPLE OF A PAGE-LEVEL LOCK



A row-level lock

- is much less restrictive
- 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.
- Modern DBMSs automatically escalate a lock from a row level to a page level when the application session requests multiple locks on the same page. Figure 10.6 illustrates the use of a row-level lock.

FIGURE 10.6 AN EXAMPLE OF A ROW-LEVEL LOCK



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.
- clearly yields the most flexible multiuser data access
- it is rarely USED
  - extremely high level of computer overhead
  - row-level lock is much more useful in practice.

LOCK TYPES

- Binary Lock
  - A **binary lock** has only two states: locked (1) or unlocked (0).
  - If an object such as 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 accessed data item. Such operations are automatically managed and scheduled by the DBMS; the user does not lock or unlock data items. (Every DBMS has a default-locking mechanism. If the end user wants to override the default settings, the LOCK TABLE command and other SQL commands are available for that purpose.)
-

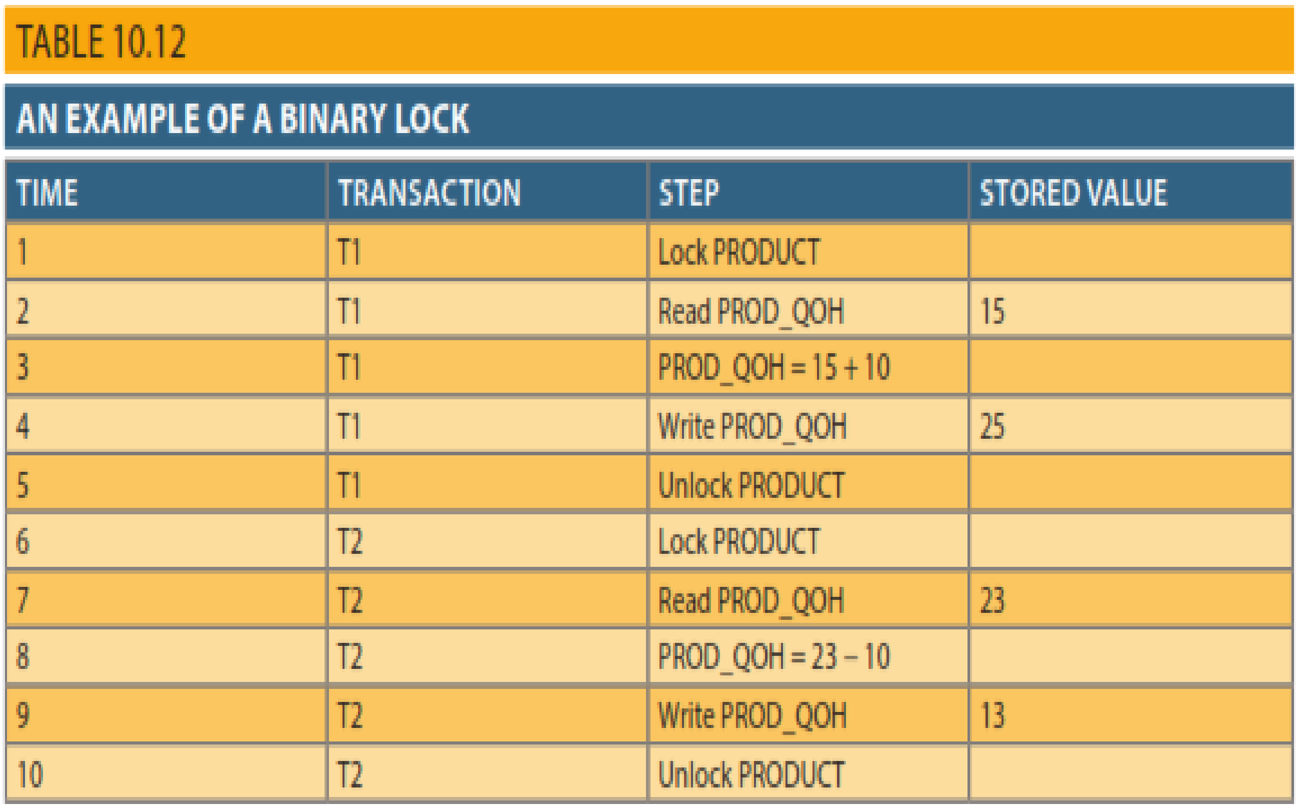


TABLE 10.12

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	

The binary locking technique is illustrated in Table 10.12, using the lost update problem you encountered in Table 10.4. Note that the lock and unlock features eliminate lost update problem

Shared/Exclusive

- An **exclusive lock** exists when access is reserved specifi- cally for the transaction that locked the object.
  - The exclusive lock must be used when the potential for conflict exists (see Table 10.11).
  - 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.

- 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.
  - Using the shared/exclusive locking concept, a lock can have three states: unlocked, shared (read), and exclusive (write).

- two transactions conflict only when at least one 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*** **(this condition is known as the mutual exclusive rule: only one transaction at a time can own an exclusive lock on an object.)**
- 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. In other words, a shared lock will always block an exclusive (write) lock; hence, decreasing transaction concurrency.

- Although the use of shared locks renders data access more efficient, 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.

Problems of locks

- 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 similar to traffic gridlock in a big city, is caused when two or more transactions wait for each other to unlock data.
- Fortunately, both problems can be managed:
  - **serializability is attained through a locking protocol known as two-phase locking,**
  - **deadlocks can be managed by using deadlock detection and prevention techniques.**

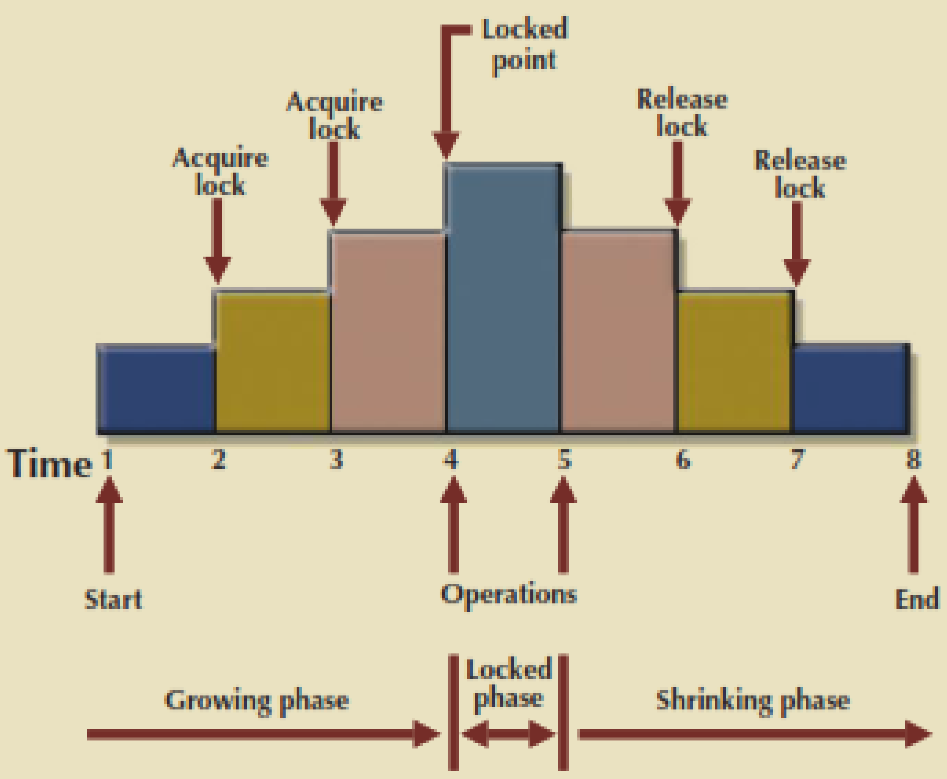
2 PHASE LOCKING

- **Two-phase locking (2PL)** defines how transactions acquire and relinquish locks.
- Two-phase locking guarantees serializability, but it does not prevent deadlocks.
- The two phases are:
  - GROWING PHASE AND SHRINKING PHASE

- A growing phase, in which a transaction acquires all required locks without unlock- ing any data. Once all locks have been acquired, the transaction is in its locked point.
- A shrinking phase, in which a transaction releases all locks and cannot obtain a 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 is affected until all locks are obtained—that is, until the transaction is in its locked point.

- Figure 10.7 depicts the two-phase locking protocol.
- In this example, the transaction first acquires the two locks it needs. When it has the two locks, it reaches its locked point.
- Next, the data is modified to conform to the trans-action's requirements.
- Finally, the transaction is completed as it releases all of the locks it acquired in the first phase.
- <u>Two-phase locking :disadvantages</u>
  - <u>increases the transaction processing cost</u>
  - <u>cause additional undesirable effects, such as deadlocks.</u>

FIGURE 10.7 TWO-PHASE LOCKING PROTOCOL



DEADLOCK

- 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.**

- Table 10.13 demonstrates how a deadlock condition is created.
- deadlocks are possible only when one of the transactions wants to obtain an exclusive lock on a data item; no deadlock condition can exist among *shared* locks.

TABLE 10.13

HOW A DEADLOCK CONDITION IS CREATED

TIME	TRANSACTION	REPLY	LOCK STATUS	
			DATA X	DATA Y
0			Unlocked	Unlocked
1	T1:LOCK(X)	OK	Locked	Unlocked
2	T2:LOCK(Y)	OK	Locked	Locked
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
...	....	....	....	....
...	....	....	....	....
...	....	....	....	....
...	....	....	....	....

Deadlock

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 dead- lock is found, the “victim” transaction 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.

- The choice of which deadlock control method to use depends on the database envi- ronment. For example, if the probability of deadlocks is low, deadlock detection is rec- ommended. 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 avoid- ance might be employed. All current DBMSs support deadlock detection in transac- tional databases, while some DBMSs use a blend of prevention and avoidance techniques for other types of data, such as data warehouses or XML data.