1/30   1:11:40   ★ ★ ★

←   →

TM

[Transaction Management]
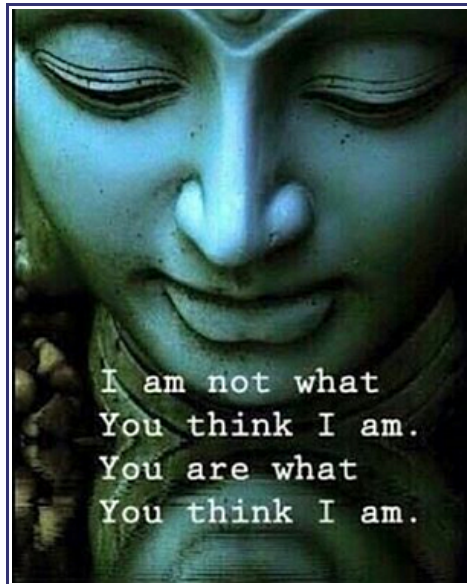
A different kind of TM :)

Ch.10

11e
Database Systems
Design, Implementation, and Management

Coronel | Morris

Chapter 10
Transaction Management and
Concurrency Control
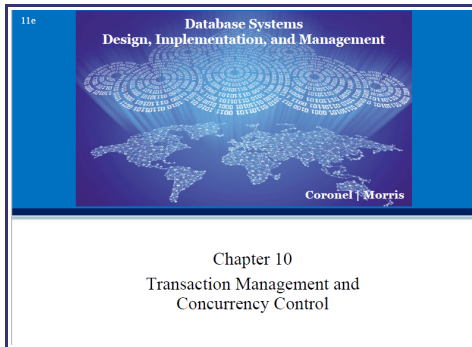
What is a 'transaction'?

Transaction
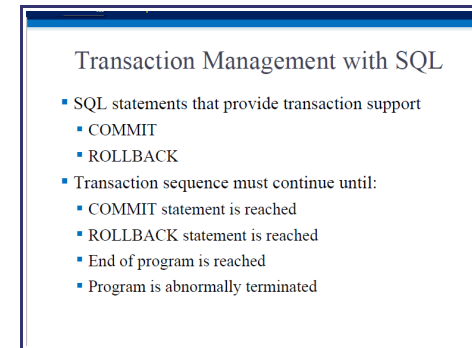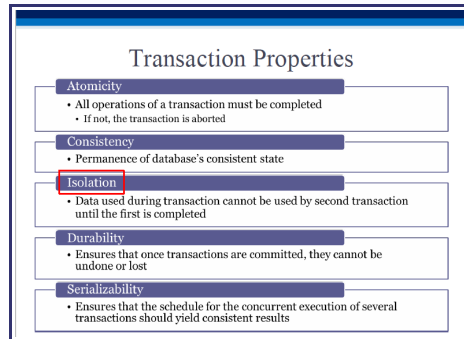
- Logical unit of work that must be entirely completed or aborted
- Consists of:
  - SELECT statement
  - Series of related UPDATE statements
  - Series of INSERT statements
  - Combination of SELECT, UPDATE, and INSERT statements

Transaction

- **Consistent database state**: All data integrity constraints are satisfied
  - Must begin with the database in a known consistent state to ensure consistency
- Formed by two or more database requests
  - **Database requests**: Equivalent of a single SQL statement in an application program or transaction
- Consists of a single SQL statement or a collection of related SQL statements

'ACID'

'ACID' (+S) is an acronym to express the desirable properties of a transaction:

Transaction Properties

Atomicity
- All operations of a transaction must be completed
  - If not, the transaction is aborted

Consistency
- Permanence of database's consistent state

Isolation
- Data used during transaction cannot be used by second transaction until the first is completed

Durability
- Ensures that once transactions are committed, they cannot be undone or lost

Serializability
- Ensures that the schedule for the concurrent execution of several transactions should yield consistent results

Transaction Management with SQL

- SQL statements that provide transaction support
  - COMMIT
  - ROLLBACK
- Transaction sequence must continue until:
  - COMMIT statement is reached
  - ROLLBACK statement is reached
  - End of program is reached
  - Program is abnormally terminated

Tracking updates

Transaction Log

- Keeps track of all transactions that update the database
- DBMS uses the information stored in a log for:
  - Recovery requirement triggered by a ROLLBACK statement
  - A program's abnormal termination
  - A system failure

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.)

Cengage Learning © 2015

Concurrency - 'many at once'

Concurrency Control

- Coordination of the simultaneous transactions execution in a multiuser database system
- Objective - Ensures serializability of transactions in a multiuser database environment

Problems in Concurrency Control

Lost update
- Occurs in two concurrent transactions when:
  - Same data element is updated
  - One of the updates is lost

Uncommitted data
- Occurs when:
  - Two transactions are executed concurrently
  - First transaction is rolled back after the second transaction has already accessed uncommitted data

Inconsistent retrievals
- Occurs when a transaction accesses data before and after one or more other transactions finish working with such data

Consider the following pair of transactions, starting with 35 units of an item: purchase 100 units, then sell 30 units:

TRANSACTION	COMPUTATION
T1: Purchase 100 units	PROD_QOH = PROD_QOH + 100
T2: Sell 30 units	PROD_QOH = PROD_QOH – 30
Cengage Learning © 2015	

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
Cengage Learning © 2015			

If the transactions T1 and T2 happen one after another (not concurrently), the PROD_QOH value would/should be 105, as shown above.

Three different kinds of errors are possible, when interleaved transactions are not handled properly.

Error #1: lost updates

'Lost update' problem:

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
Cengage Learning © 2015			

Here, instead of 105, our QOH comes out to be 5, which is incorrect.

Consider this rollback situation:

TRANSACTION	COMPUTATION
T1: Purchase 100 units	PROD_QOH = PROD_QOH + 100 (Rolled back)
T2: Sell 30 units	PROD_QOH = PROD_QOH − 30
Cengage Learning © 2015	

Proper rollback (with sequential 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
Cengage Learning © 2015			

Here the total is 5, which is correct - the QOH goes from 35 to 135, gets rolled back to 35, after which the purchase (of 30 units) happens.

Error #2: reading uncommitted data

'Uncommitted data' problem (improper rollback):

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

Cengage Learning © 2015

Here, the total comes out to 105, when it should have been 5.

Consider the following fix (of a typo made earlier - an incorrect order of 10 units was placed for 1558-QW1 by mistake, instead of ordering 1546-QQ2; now we're fixing that error):

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;

Cengage Learning © 2015

In the above, T1 is a transaction that sums up QOH; T2 is the 'correction' transaction (that fixes the incorrect purchasing).

Proper retrieval of modified data:

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

Cengage Learning © 2015

The summing transaction gets proper totals for both affected products, so the total (of 92) is correct.

Error #3: improper [premature] retrieval

'Inconsistent retrievals' problem (improper ("before update") retrieval of modified data):

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

Cengage Learning © 2015

T1 should be doing 65+13 (which would be correct), but instead does 65+23 (which makes it incorrect) - in other words, T1 retrieves the correct value (25) for 1546-QQ2, but gets the incorrect value (23) for 1558-QW1.

Inconsistent retrieval is also called a 'dirty read'.

Look up: non-repeatable reads, phantom row reads.

Concurrent, serializable schedule

The Scheduler

- Establishes the order in which the operations are executed within concurrent transactions
  - Interleaves the execution of database operations to ensure serializability and isolation of transactions
- Based on concurrent control algorithms to determine the appropriate order
- Creates serialization schedule
  - **Serializable schedule**: Interleaved execution of transactions yields the same results as the serial execution of the transactions

A serializable schedule makes concurrency immaterial (non-issue).

Concurrency Control with Locking Methods

- Locking methods - Facilitate isolation of data items used in concurrently executing transactions
- **Lock**: Guarantees exclusive use of a data item to a current transaction
- **Pessimistic locking**: Use of locks based on the assumption that conflict between transactions is likely
- **Lock manager**: Responsible for assigning and policing the locks used by the transactions

TRL pessimistic 'lock' situations: restrooms, RCS, Robert's Rules Of Order..

Locking: granularity

Lock Granularity

- Indicates the level of lock use
- Levels of locking
  - **Database-level lock**
  - **Table-level lock**
  - **Page-level lock**
    - **Page** or **diskpage**: Directly addressable section of a disk
  - **Row-level lock**
  - **Field-level lock**

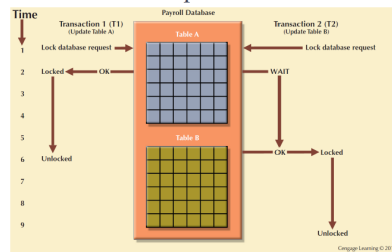### Figure 10.3 - Database-Level Locking Sequence
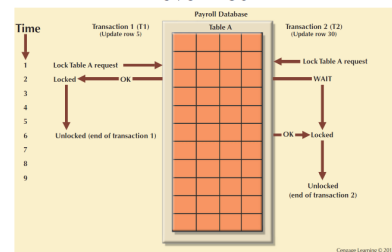

Figure 10.4 - An Example of a Table-Level Lock
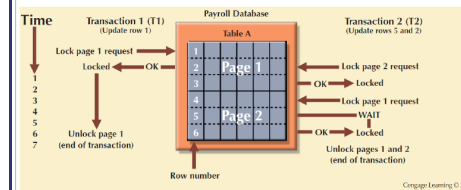
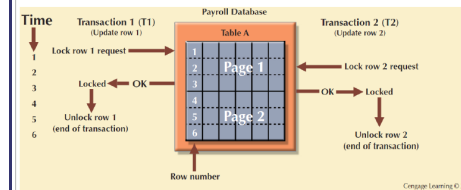

Figure 10.5 - An Example of a Page-Level Lock

Figure 10.6 - An Example of a Row-Level Lock

Locking: types [unlocked, locked_read, locked_write]

Lock Types

Binary lock
- Has two states, locked (1) and unlocked (0)
  - If an object 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

Exclusive lock
- Exists when access is reserved for the transaction that locked the object

Shared lock
- Exists when concurrent transactions are granted read access on the basis of a common lock

Three lock states

- Using the shared/exclusive concept, there are THREE lock states: unlocked, shared (read), exclusive (write)

Read(shared), write(exclusive) lock

Shared lock

- Issued when a transaction wants to READ data, and no exclusive lock is held (on a data item)

Exclusive lock

- Issued when a transaction wants to WRITE data, and no lock is held (on a data item)

'2PL'

Two-Phase Locking (2PL)

- Defines how transactions acquire and relinquish locks
- Guarantees serializability but does not prevent deadlocks
- Phases
  - Growing phase - Transaction acquires all required locks without unlocking any data
  - Shrinking phase - Transaction releases all locks and cannot obtain any new lock

The 2PL type that we are discussing, where a transaction acquires all the locks it needs before processing starts, is called 'Conservative' 2PL (or 'Static' 2PL).

2PL [cont'd]

Two-Phase Locking (2PL)

- Governing 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

Once all the locks are acquired, a transaction can proceed 'smoothly', ie won't 'hang' or lead to dirty reads etc.

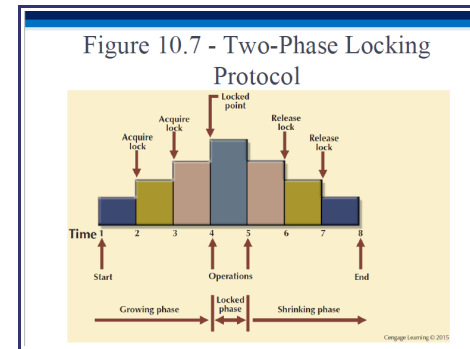As for the middle point above (unlocking can't precede locking), here is a loose analogy:

```
OK:
{
    {
        {
            {
                {
                    ....
                }
            }
        }
    }
}


Not OK (OK in programming, though!):
{
    {
    }
    {
      {
        {
        }
      }
    }
}
```

2PL [cont'd]

Figure 10.7 - Two-Phase Locking Protocol

Deadlocks..

Deadlocks

- Occurs when two transactions wait indefinitely for each other to unlock data
  - Known as **deadly embrace**
- Control techniques
  - Deadlock prevention
  - Deadlock detection
  - Deadlock avoidance
- Choice of deadlock control method depends on database environment

- *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, 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.

Timestamping-based schemes prevent deadlocks; 2PL avoids deadlocks; detection is used in both.

How/why a deadlock occurs

Table 10.13 - How a Deadlock Condition is Created

TIME	TRANSACTION	REPLY	LOCK STATUS Data X	Data Y
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
...	....	....	....	....
...	....	....	....	....
...	....	....	....	....

Cengage Learning © 2015

In the above, we see that T1 locks X, T2 locks Y, then deadlock occurs because T1 wants Y and T2 wants X. Maybe you are thinking - why can't each of them release what they are holding (since they might be done with it), and grab what they want next (ie. T1 would release X, T2 would release Y)? BECAUSE THEY CAN'T - they NEED to access what the other has, BEFORE they can release what they have. Eg. T1 might need to read Y that is locked by T2, in order to update its X; T2 might need T1's X to use in an expression to compare with its Y. **They need access to each other's resources, **before** they can release their own! That is what causes deadlocking.**

Note that more than two transactions can become deadlocked as well, on account of 'circular' (cyclical) waiting.

Here's a humorous take on deadlocking that can occur in human interactions.

Deadlock occurrence in 2PL

Earlier we noted that the 2PL scheme cannot prevent deadlock creation. Here is an example of how a deadlock could occur (at the end of step 5):

An important and unfortunate property of 2PL schedulers is that they are subject to *deadlocks*. For example, suppose a 2PL scheduler is processing transactions T_1 and T_2

$$T_1: r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_2: w_2[y] \rightarrow w_2[x] \rightarrow c_2$$

and consider the following sequence of events:

1. Initially, neither transaction holds any locks.
2. The scheduler receives $r_1[x]$ from the TM. It sets $rl_1[x]$ and submits $r_1[x]$ to the DM.
3. The scheduler receives $w_2[y]$ from the TM. It sets $wl_2[y]$ and submits $w_2[y]$ to the DM.
4. The scheduler receives $w_2[x]$ from the TM. The scheduler does not set $wl_2[x]$ because it conflicts with $rl_1[x]$ which is already set. Thus $w_2[x]$ is delayed.
5. The scheduler receives $w_1[y]$ from the TM. As in (4), $w_1[y]$ must be delayed.

Time Stamping

- Assigns global, unique time stamp to each transaction
  - Produces explicit order in which transactions are submitted to DBMS
- Properties
  - **Uniqueness**: Ensures no equal time stamp values exist
  - **Monotonicity**: Ensures time stamp values always increases

Here is one way to get monotonically increasing GUIDs..

Deadlock prevention

Time Stamping

- Disadvantages
  - Each value stored in the database requires two additional stamp fields
  - Increases memory needs
  - Increases the database's processing overhead
  - Demands a lot of system resources

Wait/Die and Wound/Wait
Concurrency Control Schemes

Two different schemes (Wait or Die, Wound or Wait) for requesting access.

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

Cengage Learning © 2015

- Top row: older transaction requests a lock
- Bottom row: newer transaction is requesting

Note that wound-wait is a preemptive deadlock prevention scheme, whereas wait-die is a non-preemptive one..

PS: Deadlock detection is periodically carried out by detecting cycles in waiting transactions.

Deadlock 'indifference' ['fix-it-later']