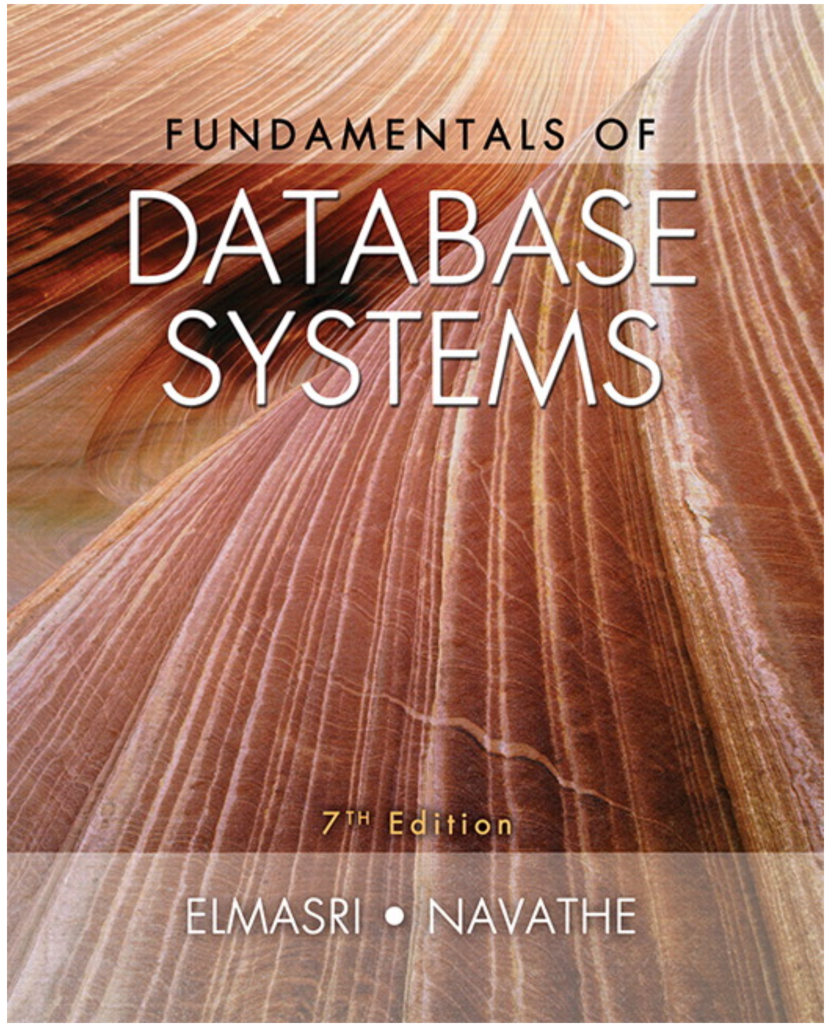
FUNDAMENTALS OF
DATABASE
SYSTEMS
7TH Edition
ELMASRI • NAVATHE

CHAPTER 21

Concurrency Control Techniques

Introduction

- Concurrency control protocols
  - Set of rules to guarantee serializability
- Two-phase locking protocols
  - Lock data items to prevent concurrent access
- Timestamp
  - Unique identifier for each transaction
- Multiversion currency control protocols
  - Use multiple versions of a data item
- Validation or certification of a transaction

21.1 Two-Phase Locking Techniques for Concurrency Control

- Lock
  - Variable associated with a data item describing status for operations that can be applied
  - One lock for each item in the database
- Binary locks
  - Two states (values)
    - Locked (1)
      - Item cannot be accessed
    - Unlocked (0)
      - Item can be accessed when requested

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Transaction requests access by issuing a lock_item(X) operation

```
lock_item(X):
B:  if LOCK(X) = 0                  (*item is unlocked*)
        then LOCK(X) ← 1            (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                    (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 21.1 Lock and unlock operations for binary locks

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock table specifies items that have locks
- Lock manager subsystem
  - Keeps track of and controls access to locks
  - Rules enforced by lock manager module
- At most one transaction can hold the lock on an item at a given time
- Binary locking too restrictive for database items

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Shared/exclusive or read/write locks
  - Read operations on the same item are not conflicting
  - Must have exclusive lock to write
  - Three locking operations
    - read_lock(X)
    - write_lock(X)
    - unlock(X)

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Figure 21.2 Locking and unlocking operations for two-mode (read/write, or shared/exclusive) locks

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock conversion
  - Transaction that already holds a lock allowed to convert the lock from one state to another
- Upgrading
  - Issue a read_lock operation then a write_lock operation
- Downgrading
  - Issue a read_lock operation after a write_lock operation

Guaranteeing Serializability by Two-Phase Locking

- Two-phase locking protocol
  - All locking operations precede the first unlock operation in the transaction
  - Phases
    - Expanding (growing) phase
      - New locks can be acquired but none can be released
      - Lock conversion upgrades must be done during this phase
    - Shrinking phase
      - Existing locks can be released but none can be acquired
      - Downgrades must be done during this phase

Figure 21.3 Transactions that do not obey two-phase locking (a) Two transactions *T*1 and *T*2 (b) Results of possible serial schedules of *T*1 and *T*2 (c) A nonserializable schedule *S* that uses locks

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

Time ↓

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Guaranteeing Serializability by Two-Phase Locking

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

Variations of Two-Phase Locking

- Basic 2PL
  - Technique described on previous slides
- Conservative (static) 2PL
  - Requires a transaction to lock all the items it accesses before the transaction begins
    - Predeclare read-set and write-set
  - Deadlock-free protocol
- Strict 2PL
  - Transaction does not release exclusive locks until after it commits or aborts

Variations of Two-Phase Locking (cont'd.)

- Rigorous 2PL
  - Transaction does not release any locks until after it commits or aborts
- Concurrency control subsystem responsible for generating read_lock and write_lock requests
- Locking generally considered to have high overhead

Dealing with Deadlock and Starvation

- Deadlock
  - Occurs when each transaction T in a set is waiting for some item locked by some other transaction *T'*
  - Both transactions stuck in a waiting queue

(a)

Time ↓

T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

(b)

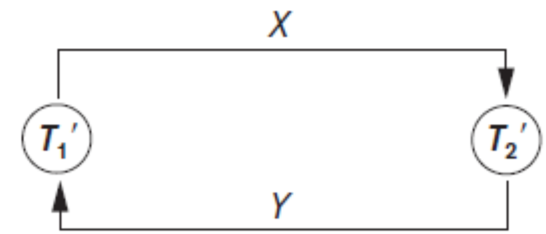

Figure 21.5 Illustrating the deadlock problem (a) A partial schedule of *T*1′ and *T*2′ that is in a state of deadlock (b) A wait-for graph for the partial schedule in (a)

Dealing with Deadlock and Starvation (cont'd.)

- Deadlock prevention protocols
  - Every transaction locks all items it needs in advance
  - Ordering all items in the database
    - Transaction that needs several items will lock them in that order
  - Both approaches impractical
- Protocols based on a timestamp
  - Wait-die
  - Wound-wait

Dealing with Deadlock and Starvation (cont'd.)

- No waiting algorithm
  - If transaction unable to obtain a lock, immediately aborted and restarted later
- Cautious waiting algorithm
  - Deadlock-free
- Deadlock detection
  - System checks to see if a state of deadlock exists
  - Wait-for graph

Dealing with Deadlock and Starvation (cont'd.)

- Victim selection
  - Deciding which transaction to abort in case of deadlock
- Timeouts
  - If system waits longer than a predefined time, it aborts the transaction
- Starvation
  - Occurs if a transaction cannot proceed for an indefinite period of time while other transactions continue normally
  - Solution: first-come-first-served queue

21.2 Concurrency Control Based on Timestamp Ordering

- Timestamp
  - Unique identifier assigned by the DBMS to identify a transaction
  - Assigned in the order submitted
  - Transaction start time
- Concurrency control techniques based on timestamps do not use locks
  - Deadlocks cannot occur

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Generating timestamps
  - Counter incremented each time its value is assigned to a transaction
  - Current date/time value of the system clock
    - Ensure no two timestamps are generated during the same tick of the clock
- General approach
  - Enforce equivalent serial order on the transactions based on their timestamps

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Timestamp ordering (TO)
  - Allows interleaving of transaction operations
  - Must ensure timestamp order is followed for each pair of conflicting operations
- Each database item assigned two timestamp values
  - read_TS(X)
  - write_TS(X)

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Basic TO algorithm
  - If conflicting operations detected, later operation rejected by aborting transaction that issued it
  - Schedules produced guaranteed to be conflict serializable
  - Starvation may occur
- Strict TO algorithm
  - Ensures schedules are both strict and conflict serializable

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Thomas's write rule
  - Modification of basic TO algorithm
  - Does not enforce conflict serializability
  - Rejects fewer write operations by modifying checks for write_item(X) operation

21.3 Multiversion Concurrency Control Techniques

- Several versions of an item are kept by a system
- Some read operations that would be rejected in other techniques can be accepted by reading an older version of the item
  - Maintains serializability
- More storage is needed
- Multiversion currency control scheme types
  - Based on timestamp ordering
  - Based on two-phase locking
  - Validation and snapshot isolation techniques

Multiversion Concurrency Control Techniques (cont'd.)

- Multiversion technique based on timestamp ordering
  - Two timestamps associated with each version are kept
    - read_TS(X_i)
    - write_TS(X_i)

Multiversion Concurrency Control Techniques (cont'd.)

- Multiversion two-phase locking using certify locks
  - Three locking modes: read, write, and certify

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 21.6 Lock compatibility tables (a) Lock compatibility table for read/write locking scheme (b) Lock compatibility table for read/write/certify locking scheme

21.4 Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

- Optimistic techniques
  - Also called validation or certification techniques
  - No checking is done while the transaction is executing
  - Updates not applied directly to the database until finished transaction is validated
    - All updates applied to local copies of data items
  - Validation phase checks whether any of transaction's updates violate serializability
    - Transaction committed or aborted based on result

Concurrency Control Based on Snapshot Isolation

- Transaction sees data items based on committed values of the items in the database snapshot
  - Does not see updates that occur after transaction starts
- Read operations do not require read locks
  - Write operations require write locks
- Temporary version store keeps track of older versions of updated items
- Variation: serializable snapshot isolation (SSI)

21.5 Granularity of Data Items and Multiple Granularity Locking

- Size of data items known as granularity
  - Fine (small)
  - Coarse (large)
- Larger the data item size, lower the degree of concurrency permitted
  - Example: entire disk block locked
- Smaller the data item size, more locks required
  - Higher overhead
- Best item size depends on transaction type

Multiple Granularity Level Locking

- Lock can be requested at any level

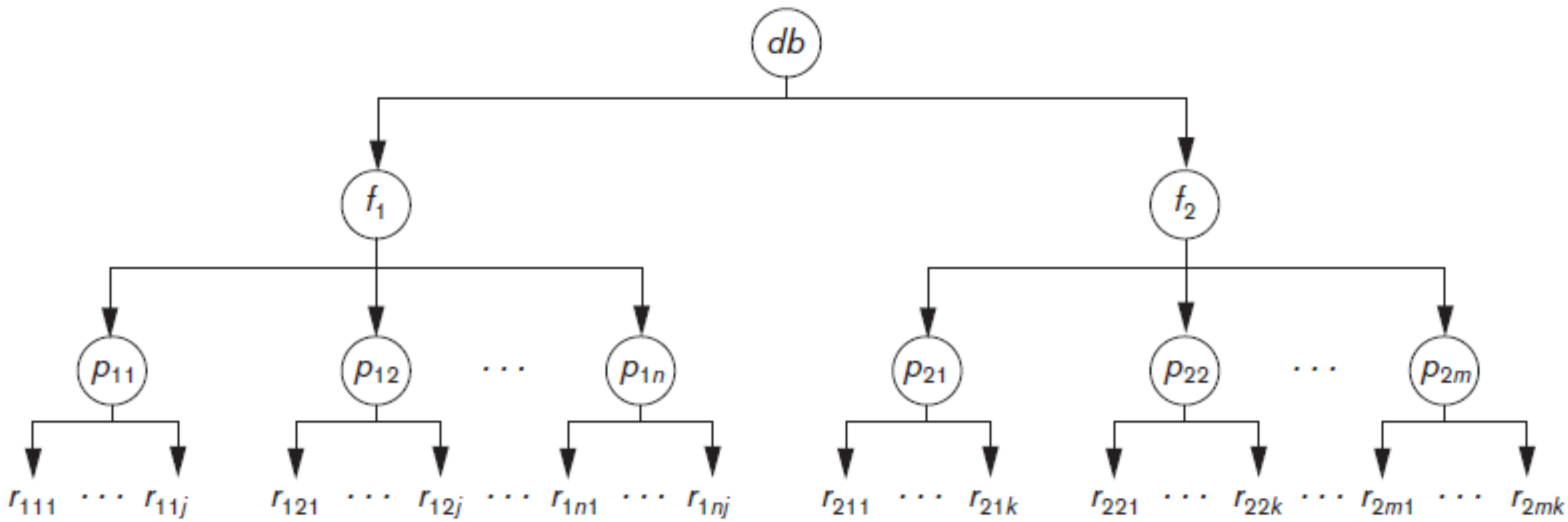

Figure 21.7 A granularity hierarchy for illustrating multiple granularity level locking

Multiple Granularity Level Locking (cont'd.)

- Intention locks are needed
  - Transaction indicates along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants
- Intention lock types
  - Intention-shared (IS)
    - Shared locks will be requested on a descendant node
  - Intention-exclusive (IX)
    - Exclusive locks will be requested

Multiple Granularity Level Locking (cont'd.)

- Intention lock types (cont'd.)
  - Shared-intension-exclusive (SIX)
    - Current node is locked in shared mode but one or more exclusive locks will be requested on a descendant node

IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 21.8 Lock compatibility matrix for multiple granularity locking

Multiple Granularity Level Locking (cont'd.)

- Multiple granularity locking (MGL) protocol rules

1. The lock compatibility (based on Figure 21.8) must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node *N* can be locked by a transaction *T* in S or IS mode only if the parent node *N* is already locked by transaction *T* in either IS or IX mode.
4. A node *N* can be locked by a transaction *T* in X, IX, or SIX mode only if the parent of node *N* is already locked by transaction *T* in either IX or SIX mode.
5. A transaction *T* can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
6. A transaction *T* can unlock a node, *N*, only if none of the children of node *N* are currently locked by *T*.

21.6 Using Locks for Concurrency Control in Indexes

- Two-phase locking can be applied to B-tree and B+ -tree indexes
  - Nodes of an index correspond to disk pages
- Holding locks on index pages could cause transaction blocking
  - Other approaches must be used
- Conservative approach
  - Lock the root node in exclusive mode and then access the appropriate child node of the root

Using Locks for Concurrency Control in Indexes (cont'd.)

- Optimistic approach
  - Request and hold shared locks on nodes leading to the leaf node, with exclusive lock on the leaf
- B-link tree approach
  - Sibling nodes on the same level are linked at every level
  - Allows shared locks when requesting a page
  - Requires lock be released before accessing the child node

21.7 Other Concurrency Control Issues

- Insertion
  - When new data item is inserted, it cannot be accessed until after operation is completed
- Deletion operation on the existing data item
  - Write lock must be obtained before deletion
- Phantom problem
  - Can occur when a new record being inserted satisfies a condition that a set of records accessed by another transaction must satisfy
  - Record causing conflict not recognized by concurrency control protocol

Other Concurrency Control Issues (cont'd.)

- Interactive transactions
  - User can input a value of a data item to a transaction *T* based on some value written to the screen by transaction *T'*, which may not have committed
  - Solution approach: postpone output of transactions to the screen until committed
- Latches
  - Locks held for a short duration
  - Do not follow usual concurrency control protocol

21.8 Summary

- Concurrency control techniques
  - Two-phase locking
  - Timestamp-based ordering
  - Multiversion protocols
  - Snapshot isolation
- Data item granularity
- Locking protocols for indexes
- Phantom problem and interactive transaction issues