Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Chapter 5: Concurrency Control Techniques

Database Management Systems (CO3021)

Computer Science Program

Dr. Võ Thị Ngọc Châu

(chauvtn@hcmut.edu.vn)

Semester 1 – 2018-2019

Course outline

- Overall Introduction to Database Management Systems
- Chapter 1. Disk Storage and Basic File Structures
- Chapter 2. Indexing Structures for Files
- Chapter 3. Algorithms for Query Processing and Optimization
- Chapter 4. Introduction to Transaction Processing Concepts and Theory
- **Chapter 5. Concurrency Control Techniques**
- Chapter 6. Database Recovery Techniques

References

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 6th Edition, Pearson- Addison Wesley, 2011.
  - ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***
- [2] H. G. Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice-Hall, 2000.
- [3] H. G. Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice-Hall, 2002
- [4] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts –3rd Edition, McGraw-Hill, 1999.
- [Internet] …

Content

- 5.1. Purposes Of Concurrency Control
- 5.2. Two-Phase Locking
- 5.3. Concurrency Control Based On Timestamp Ordering
- 5.4. Multiversion Concurrency Control Techniques
- 5.5. Validation (Optimistic) Concurrency Control Techniques
- 5.6. Granularity of Data Items and Multiple Granularity Locking Technique
- 5.7. Using Locks for Concurrency Control in Indexes
- 5.8. Other Concurrency Control Issues

5.1. Purposes Of Concurrency Control

- Concurrency control is needed:
  - To enforce ***isolation*** (through mutual exclusion) among conflicting transactions;
  - To preserve ***database consistency*** through consistency preserving execution of transactions;
  - To resolve ***read-write and write-write conflicts***.
- For example:
  - In concurrent execution environment if T_1 conflicts with T_2 over a data item A, then the existing concurrency control decides if T_1 or T_2 should get the A and if the other transaction is rolled-back or waits.

Concurrency control techniques

- Concurrency control techniques for those purposes
  - The two-phase locking technique of locking data items to prevent multiple transactions from accessing the items concurrently
    - Multiversion concurrency control
  - Concurrency control protocols that use timestamps
    - Multiversion concurrency control
  - A protocol based on the concept of validation or certification of a transaction after it executes its operations
  - Multiple granularity level two-phase locking protocol

5.2. Two-Phase Locking

Two-Phase Locking Techniques

Locking is an operation which secures (a) permission to *read* or (b) permission to *write* a data item for a transaction.

Example: *Lock*(*X*). Data item *X* is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item.

Example: *Unlock*(*X*). Data item *X* is made available to all other transactions.

Lock and *Unlock* are atomic operations.

5.2. Two-Phase Locking

- Based on the concept of *locking* data items to guarantee serializability of transaction schedules
- Two phases: expanding or growing (first) phase, shrinking (second) phase
  - A transaction is said to follow the *two-phase locking* protocol (i.e. well-formed) if *all locking* operations precede the *first unlock* operation in the transaction.
- Problems with locks: deadlock, starvation

Transaction T that follows the *two-phase locking* protocol

Locking data items
Unlocking data items

Expanding/growing (first) phase

Shrinking (second) phase

5.2. Two-Phase Locking

- A ***lock*** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
  - Used as a means of synchronizing the access by concurrent transactions to the database items
- Types of locks: binary locks, shared/exclusive (or read/write) locks, certify locks
  - System lock tables
  - Lock compatibility tables
  - Conversion of locks

5.2. Two-Phase Locking

- Binary locks
  - Two states (values): locked (1) & unlocked (0)
  - If the value of the lock on data item X is **1**, i.e. LOCK(X), then item X *cannot be accessed* by a database operation (read/write) that requests the item.
  - If the value of the lock on X is **0**, then the item can be accessed when requested.
  - → A binary lock enforces *mutual exclusion* on the data item.
    - → At most one transaction can hold the lock on a data item.
- Two operations used with binary locking: lock_item, unlock_item

5.2. Two-Phase Locking

```
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.

[1], pp. 783.

5.2. Two-Phase Locking

- Every transaction must obey the following rules in the *binary locking scheme*:
  - 1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
  - 2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
  - 3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.
  - 4. A transaction will not issue an unlock_item(X) operation unless it already holds the lock on item X.

5.2. Two-Phase Locking

- Shared/Exclusive (Read/Write) locks
  - Three states: read-locked, write-locked, unlocked
    - Read-locked: *share*-locked → other transactions are allowed to read the item.
    - Write-locked: *exclusive*-locked → a single transaction exclusively holds the lock on the item.
    - → *Less restrictive* than the binary locking scheme
- Three locking operations: read_lock(X), write_lock(X), unlock(X)

5.2. Two-Phase Locking

```
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;
```

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

[1], pp. 785.

5.2. Two-Phase Locking

```
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;
```

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

[1], pp. 785.

5.2. Two-Phase Locking

```
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.

[1], pp. 785.

5.2. Two-Phase Locking

- Every transaction must obey the following rules in the *shared/exclusive locking scheme*:
  - 1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
  - 2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
  - 3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
  - 4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
  - 5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X.
  - 6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Note: Rules 4 & 5 may be relaxed for *lock conversion*.

5.2. Two-Phase Locking

- Conversion of locks
  - A transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.
  - Upgrade: read_lock(X) → write_lock(X)
    - None of other transactions holds a lock on X.
  - Downgrade: write_lock(X) → read_lock(X)
  - When upgrading and downgrading of locks is used, the lock table must include *transaction identifiers* in the record structure for each lock to store the information on which transactions hold locks on the item.

5.2. Two-Phase Locking

- Two-phase locking protocols
  - A transaction is said to follow the two-phase locking protocol if *all locking* operations (read_lock, write_lock) precede the *first unlock* operation in the transaction.
  - Such a transaction can be divided into two phases: an ***expanding or growing (first) phase***, during which new locks on items can be acquired but none can be released; a ***shrinking (second) phase***, during which existing locks can be released but no new locks can be acquired.
  - If lock conversion is allowed, then upgrading of locks must be done during the expanding phase, and downgrading of locks must be done in the shrinking phase.
  - If *every* transaction in a schedule follows the *two-phase locking* protocol, the schedule is *guaranteed to be serializable*.
  - → When to lock data items?
  - → When to unlock data items?

5.2. Two-Phase Locking

- Two-phase locking techniques: essential components
  - Lock Manager: Managing locks on data items.
  - Lock Table: a table that stores the identity of transaction locking (the data item, lock mode and pointer to the next data item locked). One simple way to implement a lock table is through linked lists.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

5.2. Two-Phase Locking

(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

Figure 21.3 Transactions that do ***not*** obey two-phase locking

[1], pp. 787.

5.2. Two-Phase Locking

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

Result of schedule S: X=50, Y=50 (nonserializable)

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);	

5.2. Two-Phase Locking

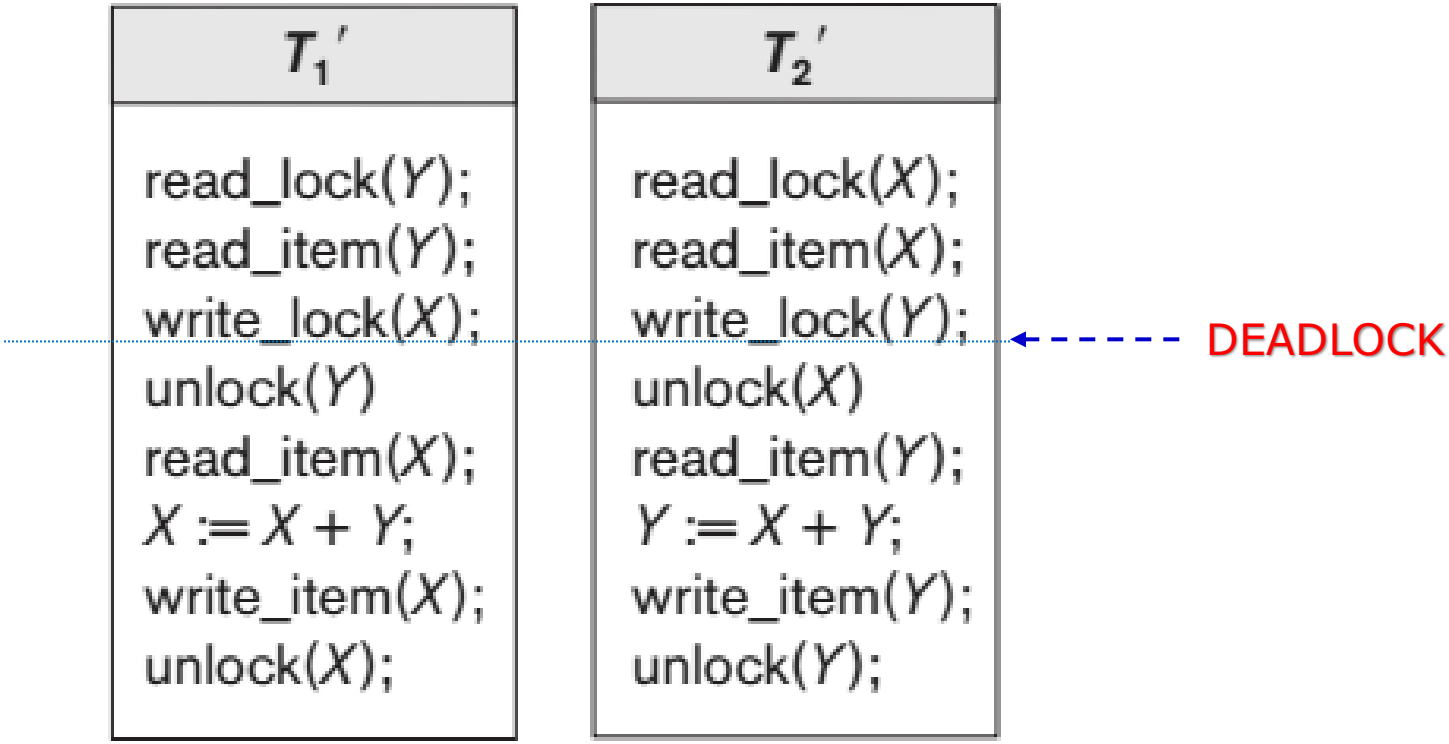


Figure 21.4 Transactions T_1' and T_2' which *follow* the two-phase locking protocol. Note that they can produce a *deadlock*.

[1], pp. 788.

5.2. Two-Phase Locking

- Variations of two-phase locking (2PL)
  - Basic 2PL
  - Conservative 2PL (static 2PL)
    - A transaction locks all the items it accesses *before* the transaction begin execution, by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any item. This is a deadlock-free protocol.
  - Strict 2PL
    - A transaction T does not release any of its *exclusive (write)* locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. This is not deadlock-free.
  - Rigorous 2PL
    - A transaction T does not release any of its locks (*exclusive or shared*) until *after* it commits or aborts, leading to a strict schedule for recoverability but easier for implementation. This is not deadlock-free.

5.2. Two-Phase Locking

- Deadlock
  - Deadlock occurs when *each* transaction T in a set of *two or more transactions* is waiting for some item that is locked by some other transaction T' in the set.
  - Deadlock *prevention*
    - Transactions are long; each transaction uses many items; the transaction load is quite heavy.
  - Deadlock *detection*
    - There will be little interference among the transactions – that is, different transactions will rarely access the same items at the same time; transactions are short; each transaction locks only a few items; the transaction load is light.

5.2. Two-Phase Locking

- Deadlock prevention
  - Conservative 2PL
  - Timestamp-based deadlock prevention schemes: wait-die & wound-wait
    - Wait-die: if transaction T_i tries to lock an item X and is older than transaction T_j that currently locks X with a conflicting lock, T_i is allowed to *wait*; otherwise T_i *dies* and is restarted *with the same timestamp*.
    - Wound-wait: If T_i is older than T_j, T_i *wounds* T_j, abort T_j and restart it later *with the same timestamp*; otherwise, T_i is allowed to *wait*.
  - Deadlock prevention schemes that do not require timestamps: no-waiting & cautious-waiting
    - No-waiting: if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.
    - Cautious-waiting: if T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

5.2. Two-Phase Locking

- Deadlock detection
  - The system checks if a state of deadlock actually exists.
  - If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted.
  - Choosing which transactions to abort is known as *victim selection*.
    - Avoid selecting transactions that have been running for a long time and that have performed many updates
  - A simple way to detect a state of deadlock is for the system to construct and maintain a *wait-for graph*.

5.2. Two-Phase Locking

- Deadlock detection
  - Construct and maintain a ***wait-for graph***
    - One node is created for each transaction that is currently executing.
    - A directed edge ($T_i \rightarrow T_j$) is created whenever a transaction T_i is waiting to lock and item X that is currently locked by a transaction T_j.
    - The directed edge ($T_i \rightarrow T_j$) is dropped when T_j releases the lock(s) on the items that T_i was waiting for.
    - → A state of deadlock *if and only if* the wait-for graph has a *cycle*.

5.2. Two-Phase Locking

(a)

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

Time

(b)

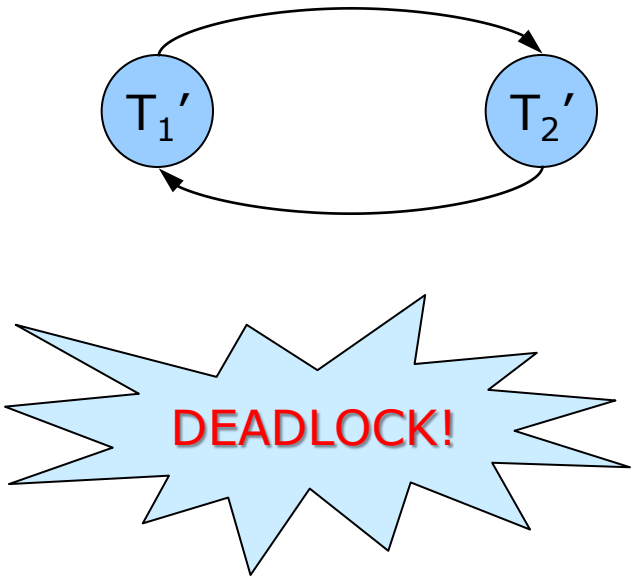


Figure 21.5 Illustrating the deadlock problem.

(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).

[1], pp. 790.

5.2. Two-Phase Locking

- Starvation
  - A transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
    - The waiting scheme for locked items is unfair, giving priority to some transactions over others.
    - Victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
  - Solutions:
    - A fair waiting scheme
      - Using a first-come-first-served queue; transactions are enabled to lock an item in the order in which they originally requested the lock.
    - Priority-based schemes
    - Wait-die and wound-wait schemes

5.3. Concurrency Control Based On Timestamp Ordering

- Guarantees serializability by using transaction timestamps to order transaction execution for an equivalent serial schedule
  - ***Timestamp ordering***: order the transactions based on their timestamps
  - → Ensure: for each item accessed by *conflicting operations*, the order in which the item is accessed does not violate the *serializability order*.
  - → Deadlock-free

5.3. Concurrency Control Based On Timestamp Ordering

- Transaction timestamp - TS(T): a unique identifier to identify a transaction T, assigned in the order in which the transaction is submitted to the system
- Read timestamp of item X – read_TS(X): the *largest* timestamp among all the timestamps of transactions that have successfully read item X – that is, read_TS(X) = TS(T), where T is the *youngest* transaction that has read X successfully
- Write timestamp of item X – write_TS(X): the *largest* of all the timestamps of transactions that have successfully written item X – that is, write_TS(X) = TS(T), where T is the *youngest* transaction that has written X successfully.

5.3. Concurrency Control Based On Timestamp Ordering

- The basic timestamp ordering algorithm
  - Check if conflicting operations violate the timestamp ordering
  - → Schedules are guaranteed to be conflict serializable.
  - 1. Transaction T issues a write_item(X) operation
    - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation because some younger transaction with a timestamp greater than TS(T) – and hence, after T in the timestamp ordering – has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
    - Otherwise, execute the write_item(X) operation of T and set write_TS(X) to TS(T).
  - 2. Transaction T issues a read_item(X) operation
    - If write_TS(X) > TS(T), then abort and roll back T and reject the operation because some younger transaction with timestamp greater than TS(T) – and hence, after T in the timestamp ordering – has already written the value of item X before T had a chance to read X.
    - Otherwise, execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

5.3. Concurrency Control Based On Timestamp Ordering

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)	Read_item(X)		
	Write_item(X)	Write_item(X)		
	Read_item(Y)			
	Write_item(Y)			

Serial schedule S of transactions T1 & T2 based on *timestamp*:

S = T1; T2 = r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(X)

5.3. Concurrency Control Based On Timestamp Ordering

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)		Read_TS = 1	
		Read_item(X)	Read_TS = 2	
	Write_item(X)		Reject!	
	Read_item(Y)			
		Write_item(X)	Write_TS = 2	
	Write_item(Y)			

Write_item(X) of T1 is *rejected*, T1 is *aborted*: Read_TS(X) > TS(T1)

Schedule C (Figure 20.5.c) is *non-serializable*.

5.3. Concurrency Control Based On Timestamp Ordering

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)		Read_TS = 1	
	Write_item(X)		Write_TS = 1	
		Read_item(X)	Read_TS = 2	
		Write_item(X)	Write_TS = 2	
	Read_item(Y)			Read_TS = 1
	Write_item(Y)			Write_TS = 1

Schedule D (Figure 20.5.d) is *conflict serializable*.

5.3. Concurrency Control Based On Timestamp Ordering

- Strict timestamp ordering
  - Ensure: the schedules are both *strict* (for easy recoverability) and *conflict serializable*
  - A transaction T that issues a *read_item(X)* or *write_item(X)* such that TS(T) > write_TS(X) has its read or write operation ***delayed*** until the transaction T' that wrote the value of X (hence, TS(T') = write_TS(X)) has committed or aborted.

5.3. Concurrency Control Based On Timestamp Ordering

- Thomas's Write Rule
  - A modification of the basic timestamp ordering algorithm that does *not* enforce conflict serializability; but rejects fewer write operations
  - Checks for the write_item(X) operation:
    - 1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
    - 2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T) – and hence, after T in the timestamp ordering – has already written the value of X.
    - 3. Otherwise, execute the write_item(X) operation of T and set write_TS(X) to TS(T).

5.4. Multiversion concurrency control techniques

- Several *versions* of an item are maintained.
  - When a transaction writes an item, it writes a new version (new value) and the old version (old value) of the item is maintained.
- When a transaction requires access to an item, an *appropriate version* is chosen to maintain the serializability of the currently executing schedule.
- Some *read* operations can be accepted by reading an older version of the item to maintain serializability.

5.4. Multiversion concurrency control techniques

- Multiversion technique based on *timestamp ordering*
  - Several versions X_1, X_2, …, X_k of each data item X are maintained.
    - For each version, the value of version X_i and the two timestamps are kept:
      - Read_TS(X_i): the *largest* of all the timestamps of transactions that have successfully read version X_i
      - Write_TS(X_i): the timestamp of the transaction that wrote the value of version X_i
  - Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T).

5.4. Multiversion concurrency control techniques

- Multiversion technique based on *timestamp ordering*
  - To ensure *serializability*, the following two rules are used:
    - 1. If transaction T issues a *write_item(X)* operation, and version **i** of X has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).
    - 2. If transaction T issues a *read_item(X)* operation, find the version **i** of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

5.4. Multiversion concurrency control techniques

- Multiversion technique based on *timestamp ordering*

T1	T2	T3	X_0	Y_0	Z_0
TS = 20	TS = 25	TS = 15	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
R1(X)	R2(Z)	R3(Y)			
W1(X)	R2(Y)	R3(Z)			
R1(Y)	W2(Y)	W3(Y)			
W1(Y)	R2(X)	W3(Z)			
	W2(X)				

Serial schedule S of transactions T1, T2, & T3 based on *timestamp*:

S = T3; T1; T2

5.4. Multiversion concurrency control techniques

T1	T2	T3	X_0	Y_0	Z_0	X_{20}	Y_{15}	Z_{15}
TS = 20	TS = 25	TS = 15	R=0 W=0	R=0 W=0	R=0 W=0	R=20 W=20	R=15 W=15	R=15 W=15
		R3(Y)		R=15				
		R3(Z)			R=15			
R1(X)			R=20					
W1(X)						created		
		W3(Y)					created	
		W3(Z)						created
	R2(Z)							R=25
R1(Y)							R=20	
W1(Y)								
	R2(Y)							
	W2(Y)							
	R2(X)							
	W2(X)							

What happens with the W1(Y) operation of transaction T1?

5.4. Multiversion concurrency control techniques

T1	T2	T3	X_0	Y_0	Z_0	X_{20}	Y_{15}	Z_{15}	Y_{20}	Y_{25}	X_{25}
TS = 20	TS = 25	TS = 15	R=0 W=0	R=0 W=0	R=0 W=0	R=20 W=20	R=15 W=15	R=15 W=15	R=20 W=20	R=25 W=25	R=25 W=25
		R3(Y)		R=15							
		R3(Z)			R=15						
R1(X)			R=20								
W1(X)						created					
		W3(Y)					created				
		W3(Z)						created			
	R2(Z)							R=25			
R1(Y)							R=20				
W1(Y)									created		
	R2(Y)								R=25		
	W2(Y)									created	
	R2(X)					R=25					
	W2(X)										created

What happens with the W1(Y) operation of transaction T1?

5.4. Multiversion concurrency control techniques

Original schedule

T1	T2	T3	T4	A
TS=150	TS=200	TS=175	TS=225	Read_TS=0; Write_TS=0
$R_1(A)$				
$W_1(A)$				
	$R_2(A)$			
	$W_2(A)$			
		$R_3(A)$		
			$R_4(A)$	

Your turn: perform concurrency control on this schedule using the traditional timestamp ordering technique and then the multiversion one.

5.4. Multiversion concurrency control techniques

Original schedule

T1	T2	T3	T4	A
TS=150	TS=200	TS=175	TS=225	Read_TS=0; Write_TS=0
$R_1(A)$				Read_TS = 150
$W_1(A)$				Write_TS = 150
	$R_2(A)$			Read_TS = 200
	$W_2(A)$			Write_TS = 200
		$R_3(A)$		*(abort T3)*
			$R_4(A)$	Read_TS = 225

The traditional timestamp ordering technique

5.4. Multiversion concurrency control techniques

Original schedule

T1	T2	T3	T4	A	A_{T1}	A_{T2}
TS=150	TS=200	TS=175	TS=225	Read_TS=0; Write_TS=0	Read_TS = 150 Write_TS = 150	Read_TS = 200 Write_TS = 200
$R_1(A)$				Read_TS = 150		
$W_1(A)$					Created	
	$R_2(A)$				Read_TS = 200	
	$W_2(A)$					Created
		$R_3(A)$			T3 reads	
			$R_4(A)$			Read_TS = 225

The multiversion timestamp ordering technique

5.4. Multiversion concurrency control techniques

Original schedule

T1	T2	T3	A	B	C
TS=200	TS=150	TS=175	Read_TS=0; Write_TS=0	Read_TS=0; Write_TS=0	Read_TS=0; Write_TS=0
$R_1(B)$					
	$R_2(A)$				
		$R_3(C)$			
	$W_2(A)$				
$W_1(B)$					
$W_1(A)$					
	$W_2(C)$				
		$W_3(A)$			

Your turn: perform concurrency control on this schedule using the traditional timestamp ordering technique and then the multiversion one.

5.4. Multiversion concurrency control techniques

Original schedule

T1	T2	T3	A	B	C
TS=200	TS=150	TS=175	Read_TS=0; Write_TS=0	Read_TS=0; Write_TS=0	Read_TS=0; Write_TS=0
$R_1(B)$				Read_TS = 200	
	$R_2(A)$		Read_TS = 150		
		$R_3(C)$			Read_TS = 175
	$W_2(A)$		Write_TS = 150		
$W_1(B)$				Write_TS = 200	
$W_1(A)$			Write_TS = 200		
	$W_2(C)$				(*abort T2*)
		$W_3(A)$	(*abort T3*)		

Your turn: perform concurrency control on this schedule using the traditional timestamp ordering technique

5.4. Multiversion concurrency control techniques

Original schedule

T1	T2	T3	A	A_{T2}	A_{T1}	A_{T3}	B	B_{T1}	C
TS = 200	TS = 150	TS = 175	Read_TS= 0; Write_TS= 0	Read_TS = 150; Write_TS = 150	Read_TS = 200; Write_TS = 200	Read_TS = 175; Write_TS = 175	Read_TS=0 ; Write_TS= 0	Read_TS = 200; Write_TS = 200	Read_TS= 0; Write_TS= 0
$R_1(B)$							Read_TS = 200		
	$R_2(A)$		Read_TS = 150						
		$R_3(C)$							Read_TS = 175
	$W_2(A)$			Created					
$W_1(B)$								Created	
$W_1(A)$					Created				
	$W_2(C)$								(*abort T2*)
		$W_3(A)$				Created			

Your turn: perform concurrency control on this schedule using the multiversion timestamp ordering technique

5.4. Multiversion concurrency control techniques

- Multiversion *two-phase locking* using certify locks
  - A multiple-mode locking scheme
    - Three locking modes for an item: read, write, certify
    - State of LOCK(X) for an item X: read-locked, write-locked, certify-locked, or unlocked

(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) A compatibility table for read/write locking scheme

(b) A compatibility table for read/write/certify locking scheme

[1], pp. 797.

5.4. Multiversion concurrency control techniques

- Multiversion *two-phase locking* using certify locks
  - In the standard 2PL scheme, once a transaction obtains a write lock on an item X, no other transactions can access X.
  - In the multiversion 2PL scheme, other transactions T' are allowed to read an item X while a single transaction T holds a write lock on X.
    - Two versions for each item X: one version written by some committed transaction; another version X' created by transaction T that acquires a write lock on X
    - Other transactions (different from T) can continue to read the committed version of X while T holds the write lock.
    - Once T is ready to commit, T must obtain a certify lock on all items that it currently holds write locks on before it can commit.
    - Once the certify locks are acquired, the committed version X of the data item is set to the value of version X', version X' is discarded, and the certify locks are then released.
    - → No cascading aborts; but deadlocks may occur with lock upgradings.

5.4. Multiversion concurrency control techniques

Transactions T1′ and T2′

T1′	T2′	X	Y
Read_item(Y)	Read_item(X)		
Read_item(X)	Read_item(Y)		
Write_item(X)	Write_item(Y)		

Transactions T1′ and T2′ that follow the traditional 2PL protocol

T1′	T2′	X	Y
Read_lock(Y)	*Read_lock(X)*		
Read_item(Y)	Read_item(X)		
Read_lock(X)	*Read_lock(Y)*		
Read_item(X)	Read_item(Y)		
Write_lock(X)	*Write_lock(Y)*		
Write_item(X)	Write_item(Y)		
Unlock(Y)	*Unlock(X)*		
Unlock(X)	*Unlock(Y)*		

Multiversion *two-phase locking* using certify locks

T1′	T2′	$X_{committed}$	$Y_{committed}$	$X_{T1'}$	$Y_{T2'}$
Read_lock(Y)			Read_locked (T1′)		
Read_item(Y)			T1′ reads		
Write_lock(X)		Write_locked (T1′)			
	Read_lock(X)	Read_locked (T2′)			
	Read_item(X)	T2′ reads			
	Write_lock(Y)		Write_locked (T2′)		
Read_item(X)		T1′ reads			
Write_item(X)				created	
Certify_lock(X)		$X_{T1'} \rightarrow X_{committed}$		*discarded*	
	Read_item(Y)		T2′ reads		
	Write_item(Y)				created
	Certify_lock(Y)		$Y_{T2'} \rightarrow Y_{commited}$		*discarded*

Certify_lock(X): waiting for other transactions' release of Read_lock(X).
Certify_lock(Y): waiting for other transactions' release of Read_lock(Y).

Multiversion *two-phase locking* using certify locks

T1	T2
Read_item(X)	
	Read_item(Y)
Write_item(Y)	

Your turn: rewrite the schedule using the traditional 2PL protocol and then show it with multiversion two-phase locking using certify locks.

Multiversion *two-phase locking* using certify locks

Original schedule

T1	T2
Read_item(X)	
	Read_item(Y)
Write_item(Y)	

Transactions in traditional 2PL

T1	T2
Read_lock(X)	*Read_lock(Y)*
Read_item(X)	Read_item(Y)
Write_lock(Y)	*Unlock(Y)*
Write_item(Y)	
Unlock(X)	
Unlock(Y)	

Schedule with mutliversion 2PL using certify locks

T1	T2	$X_{committed}$	$Y_{committed}$	Y_{T2}
Read_lock(X)		Read_locked (T1)		
Read_item(X)		T1 reads		
	Read_lock(Y)		Read_locked(T2)	
	Read_item(Y)		T2 reads	
Write_lock(Y)			Write_locked(T1)	
Write_item(Y)				Created
	Unlock(Y)			
Certify_lock(Y)			$Y_{T2} \rightarrow Y_{commited}$	Discarded
Unlock(Y)				
Unlock(X)				

5.5. Validation (optimistic) concurrency control techniques

- No checking for concurrency control is done while the transaction is executing.
- Updates in the transaction are not applied directly to the database items until the transaction reaches its end.
- At the end of transaction execution, a validation phase checks whether any of the transaction's updates violate serializability.
- If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.
- "Optimistic" because the techniques assume that little interference will occur and hence that there is no need to do checking during transaction execution.

5.5. Validation (optimistic) concurrency control techniques

- There are three phases for this concurrency control protocol:
  - 1. *Read phase*: a transaction can read values of committed data items from the database. Updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
  - 2. *Validation phase*: checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
  - 3. *Write phase*: if the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

5.5. Validation (optimistic) concurrency control techniques

- In the *validation phase* for transaction T_i, the timestamp-based protocol checks that T_i does not interfere with any committed transactions or with any other transactions T_j currently in their validation phase.
  - 1. Transaction T_j completes its write phase before T_i starts its read phase.
  - 2. T_i starts its write phase after T_j completes its write phase, and the read_set of T_i has no items in common with the write_set of T_j.
  - 3. T_j completes its read phase before T_i completes its read phase; and both the read_set and write_set of T_i have no items in common with the write_set of T_j.
  - → Only if condition (1) is false, condition (2) is checked. Only if condition (2) is false, condition (3) is checked.
  - → If any one of these three conditions holds, there is no interference and T_i is validated successfully. If none, the validation of T_i fails and T_i is aborted and restarted later.

5.6. Granularity of Data Items and Multiple Granularity Locking Technique

- In this *multiple granularity* locking scheme, the granularity level (size of the data item) may be changed dynamically.
  - A field value of a database record
  - A database record
  - A disk block
  - A whole file
  - The whole database
  - → *Fine granularity* refers to small item sizes; *coarse granularity* refers to large item sizes.
  - → The larger the data item size is, the lower the degree of concurrency permitted.
  - → The smaller the data item size is, the more the number of items in the database; leading to a larger number of active locks to be handled by the lock manager.

5.6. Granularity of Data Items and Multiple Granularity Locking Technique

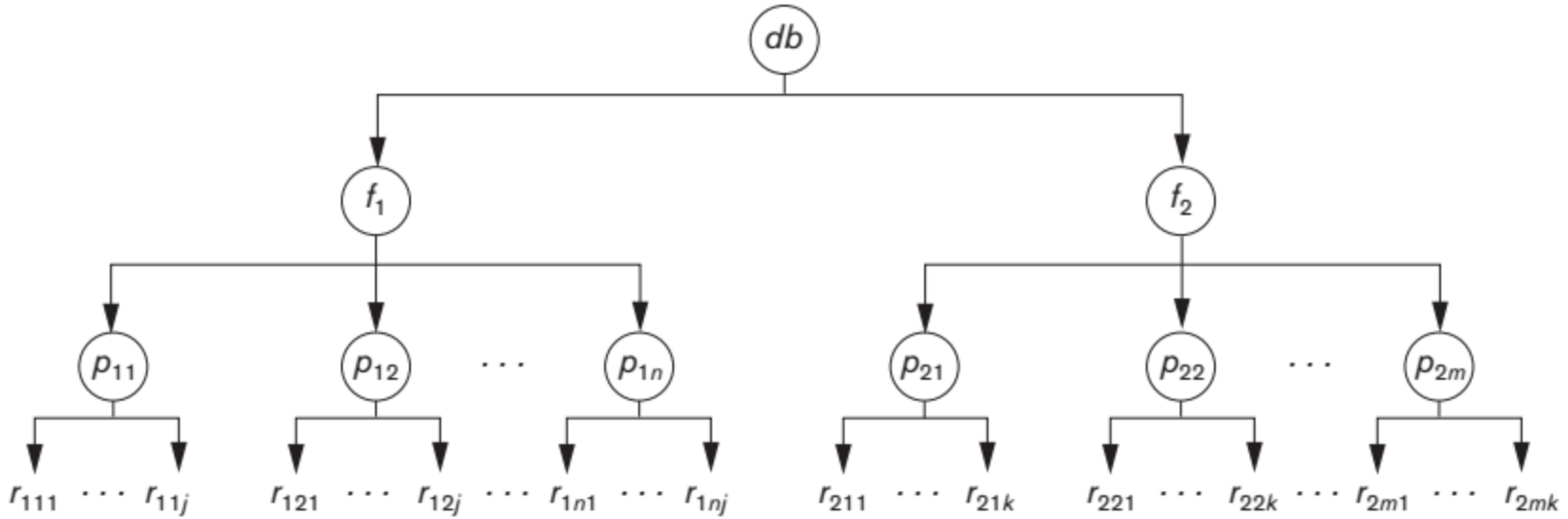
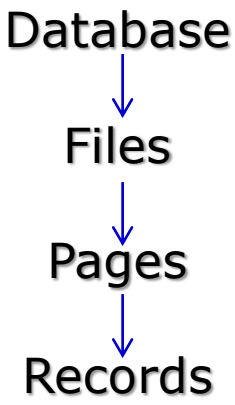


Figure 21.7 A granularity hierarchy for illustrating multiple granularity level locking. [1], pp. 802.



Where can a lock be requested?

5.6. Granularity of Data Items and Multiple Granularity Locking Technique

- Types of locks
  - Shared lock (**S**) on node N: N is locked in shared mode.
  - Exclusive lock (**X**) on node N: N is locked in exclusive mode.
  - Intention-shared lock (**IS**) on node N: a shared lock(s) will be requested on some descendant node(s) of N.
  - Intention-exclusive (**IX**) lock on node N: an exclusive lock(s) will be requested on some descendant node(s) of N.
  - Shared-intention-exclusive (**SIX**) lock on node N: N is locked in shared mode but an exclusive lock(s) will be requested on some descendant node(s).
  - → *Intention locks*: what type of lock (shared or exclusive) a transaction will require from one of the node's descendants along the path from the root to the desired node.

5.6. Granularity of Data Items and Multiple Granularity Locking Technique

	IS	IX	S	SIX	X
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 [1], pp. 803.

5.6. Granularity of Data Items and Multiple Granularity Locking Technique

- The multiple granularity locking protocol consists of the following rules:

  1. The *lock compatibility* 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 of 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 for serializability).

  6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

T1	T2	T3
IX(db)		
IX(f_1)		
	IX(db)	
		IS(db)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(r_{211})		

(cont.)

Unlock(r_{211})		
Unlock(p_{21})		
Unlock(f_2)		
		S(f_2)
	Unlock(p_{12})	
	Unlock(f_1)	
	Unlock(db)	
Unlock(r_{111})		
Unlock(p_{11})		
Unlock(f_1)		
Unlock(db)		
		Unlock(r_{11j})
		Unlock(p_{11})
		Unlock(f_1)
		Unlock(f_2)
		Unlock(db)

Using the granularity hierarchy above, consider the following three transactions in a possible serializable schedule (only the lock operations are shown):

1. T1 wants to update record r_{111} and record r_{211}.
2. T2 wants to update all records on page p_{12}.
3. T3 wants to read record r_{11j} and the entire f_2 file.

5.6. Granularity of Data Items and Multiple Granularity Locking Technique

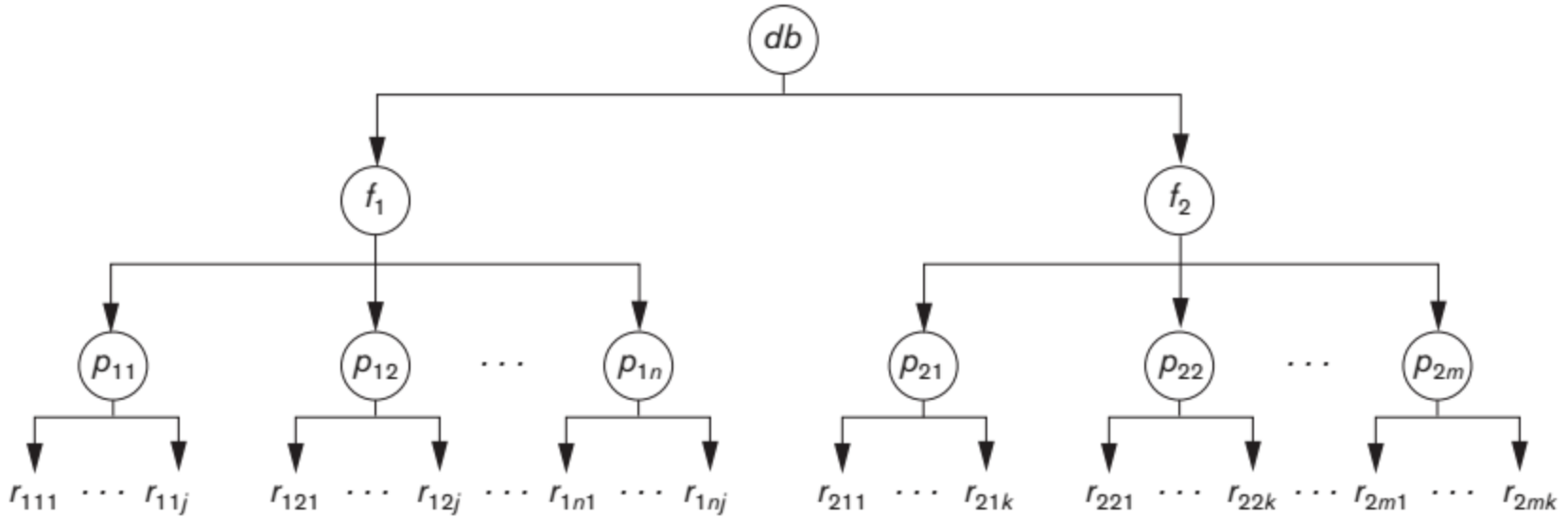


Figure 21.7 A granularity hierarchy for illustrating multiple granularity level locking. [1], pp. 802.

Your turn: Write schedules with the lock and unlock operations for transaction T4 that wants to read record r_{211}, update page p_{22}, and read page p_{12}; for transaction T5 that wants to read p_{21} and update record r_{211}.

5.7. Using Locks for Concurrency Control in Indexes

- For any data insertion, deletion, and modification, a corresponding index (B-tree, B+-tree) must be updated accordingly.
- Holding locks on index pages until the shrinking phase of two-phase locking could cause an undue amount of transaction blocking because searching an index always *starts at the root*.
- The tree structure of the index can be taken advantage of when developing a concurrency control scheme.
- **B-link tree**, a variant of B+-tree, is used.

5.8. Other Concurrency Control Issues

- Insertion, deletion, and phantom records
  - Locking and timestamps for insertion and deletion
  - Index locking for the phantom record problem
- Interactive transactions
  - A problem occurs when interactive transactions read input and write output to an interactive device before they are committed.

  → postpone output of transactions to the screen until they have committed
- Latches
  - Locks for a short duration

  → no two-phase locking

Summary

- Two-phase locking techniques
  - Basic, conservative, strict, rigorous with shared (read) and exclusive (write) locks
  - Multiversion with certify locks
  - Multiple granularity with multiple granularity level locks
  - → Deadlocks and starvations
  - → Wait-for graphs for deadlock detection
  - → Serializable schedules

Summary

- Timestamp ordering techniques
  - Basic
    - Thomas' Write Rule (not guarantee serializability)
  - Strict
  - Multiversion
  - → Serializable schedules by ordering transaction timestamps
- Other issues
  - index locking, insertion, deletion, the phantom record problem, interactive transactions, latches

Chapter 5: Concurrency Control Techniques

Check for Understandings

- 5.1. State the purposes of concurrency control. Give several examples.
- 5.2. What is a lock? Give an example.
- 5.3. What is two-phase locking? How does it ensure serializability? Give an example.
- 5.4. Differentiate the variations of the two-phase locking technique: conservative, strict, rigorous.
- 5.5. Describe deadlock and starvation problems and the approaches to dealing with them. Give an example.

Check for Understandings

- 5.6. What is a wait-for graph? How can it detect deadlock?
- 5.7. Given the following schedule. Draw the wait-for graph before and after the last action *write_lock*(*A*) of transaction T3. Will deadlock occur? Why?

T1	T2	T3	T4
read_lock(A);			
read_item(A);			
	write_lock(B);		
	write_item(B);		
read_lock(B);			
		read_lock(C);	
		read_item(C);	
	write_lock(C);		
			write_lock(B);
		write_lock(A);	

Check for Understandings

- 5.8. Describe the two-phase locking technique using certify locks. Give an example.
- 5.9. Describe the multiple granularity level two-phase locking technique. Give an example.
- 5.10. Describe the lock compatibility matrix for multiple granularity locking. Why is IS compatible with SIX? Why is IX incompatible with SIX?
- 5.11. Describe the optimistic concurrency control technique. How different is it from other concurrency control techniques?

Check for Understandings

- 5.12. Describe the timestamp ordering technique. How does it ensure serializability? Give an example.
- 5.13. Differentiate strict timestamp ordering from basic timestamp ordering.
- 5.14. Describe the multiversion technique based on timestamp ordering. Give an example.
- 5.15. What is the main difference between the multiversion techniques and their corresponding basic techniques? Give an example.

Check for Understandings

- 5.16. Describe index locking. Give an example.
- 5.17. What are latches? When are they used?
- 5.18. What is an interactive transaction? Give an example.
- 5.19. What is a phantom record? Describe its problem for concurrency control.
- 5.20. Describe two-phase locking for insertions and deletions.

Check for Understandings

- 5.21. Given two following transactions:

T1
read_item(X);
read_item(Y);
X := X+Y;
write_item(X);

T2
read_item(X);
read_item(Y);
Y := Y-X;
write_item(Y);

Form their schedule with read_lock, write_lock, and unlock operations using two-phase locking. Will deadlock occur with their execution? Explain in detail.

Check for Understandings

- 5.22. Consider the set of transactions accessing database element A. These transactions are operating under a basic timestamp-based scheduler. Explain why the transaction T3 has to be aborted. What happens if these transactions are operating under a multiversion timestamp-based scheduler?

T1	T2	T3	T4	A
150	200	175	225	RT=0 WT=0
$r_1(A)$				RT=150
$w_1(A)$				WT=150
	$r_2(A)$			RT=200
	$w_2(A)$			WT=200
		$r_3(A)$ ABORTED		
			$r_4(A)$	RT=225

Check for Understandings

- 5.23. Consider the relation Movie(title, year, length, studioName) in the database Studio.

  Transaction T1 consists of the query:

  SELECT * FROM Movie
  WHERE title = 'King Kong';

  Transaction T2 consists of the query:

  UPDATE Movie SET year = 1939
  WHERE title = 'Gone with the wind';

  Assume that there are two records in relation Movie with the title 'King Kong' and there is one record with the title 'Gone with the wind'.

  Suggest the collection of locks for this situation using multiple granularity locking.

Check for Understandings

- 5.24. Given two transactions and their schedules:

T1	T2
$r_1(X)$	$r_2(Z)$
$r_1(Z)$	$r_2(Y)$
$w_1(X)$	$w_2(Z)$
$r_1(Y)$	$r_2(X)$
$w_1(Z)$	

- Which schedule is serializable?
- Which schedule faces deadlock?

S1

T1	T2
	write_lock(Z)
	$r_2(Z)$
	read_lock(Y)
	$r_2(Y)$
	$w_2(Z)$
	read_lock(X)
	$r_2(X)$
	unlock(X)
read_lock(X)	
$r_1(X)$	
	unlock(Z)
write_lock(Z)	
$r_1(Z)$	
write_lock(X)	
$w_1(X)$	
	unlock(Y)
read_lock(Y)	
$r_1(Y)$	
$w_1(Z)$	
unlock(Z)	
unlock(Y)	
unlock(X)	

S2

T1	T2
	write_lock(Z)
	$r_2(Z)$
write_lock(X)	
$r_1(X)$	
	read_lock(Y)
	$r_2(Y)$
	$w_2(Z)$
write_lock(Z)	
	read_lock(X)
	$r_2(X)$
	unlock(X)
$r_1(Z)$	
$w_1(X)$	
	unlock(Z)
	unlock(Y)
read_lock(Y)	
$r_1(Y)$	
$w_1(Z)$	
unlock(Z)	
unlock(Y)	
unlock(X)	