Chapter 15 : Concurrency Control

Database System Concepts, 6th Ed.

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols

Concurrent Schedules

- Advantage of Concurrent Schedules

- Problem associated with the execution of concurrent schedules

- Lost Update
- Dirty Read
- Incorrect Summary

Concurrency Protocols

Goal: Generates Concurrent Schedules

Should satisfy following facts:

- Schedule is serializable : Conflict or View serializable
- Recoverable
- Cascade-less
- Increase the level of concurrency (To improve system performance)
- Reduce the protocol overhead

Lock based protocol

Lock Mechanism

exclusive (X)   for Write operation

shared (S)   for Read operation

Concurrency control Manager

lock-X instruction
lock-S instruction

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- It is a variable associated with a data item in the database and describes the status of the item w.r.t. possible operations that can be applied on to item.
- Generally there is one lock for each item.
- Binary Locks: can have two states or values: locked and unlocked (1 and 0)
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  T2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*); → T1
  read (*B*);
  unlock(*B*);
  display(*A+B*)

May lead to incorrect Summary Problem

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks

- The protocol assures serializability.
- It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither *T3* nor ng **lock-x**(*B*) causes *T4* to wait for *T* xecuting **lock-X**(*A*) causes *T3* to wait for *T4* to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a **sequence of other transactions** request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to **deadlocks**.

- Concurrency control manager can be designed to prevent starvation.

Example for Starvation

T1 **T2** **T3** **T4**

Time

S(A)

X(A) // Waiting

S(A)

C;

S(A)

C

C; // T1 resumes

The Two-Phase Locking Protocol (Cont.)

- Cascading roll-back is possible under two-phase locking.

T1	T2
X(A)	
UN(A)	
	X(A)
Aborts	

- **Solution: strict two-phase locking**:
  - Here a transaction must **hold all its exclusive locks till it commits/aborts**. After commit it must unlock.

- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Static/ Conservative 2PL protocol

- Deadlock:

T_3	T_4
lock-x (B)	
read (B)	
B := B - 50	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

T_3	T_4
lock-x (B)	
lock-x (A)	
read (B)	
B := B - 50	
write (B)	
	lock-s (A)

- Solution: Static/ Conservative 2PL protocol
- Gets Lock in Atomic Manner

- No Deadlock
- Reduces the concurrency level

Lock Conversions

To improve the concurrency level

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-R on item
    - can acquire a lock-W on item
    - can convert a lock-R to a lock-W (upgrade)
  - Second Phase:
    - can release a lock-R
    - can release a lock-W
    - can convert a lock-W to a lock-R (downgrade)

Lock Conversions

T_3
lock-x (B)
lock-x (A)
read (A)
write (A)
write (B)
read (B)

T_3
lock-s (A)
read (A)
lock-x (A)
write (A)
lock-x (B)
write (B)
lock-s (B)
read (B)

(upgrade)

(downgrade)

Automatic Acquisition of Locks

- A transaction *Ti* issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

if *Ti* has a lock on *D*
  then
    read(*D*)
  else begin
    if necessary wait until no other
      transaction has a **lock-W** on *D*
    grant *Ti* a **lock-R** on *D*;
    read(*D*)
  end

Automatic Acquisition of Locks (Cont.)

- **write**(*D*) is processed as:

  if *Ti* has a **lock-W** on *D*
    then
      write(*D*)
   else begin
      if necessary wait until no other trans. has any lock on *D*,
      if *Ti* has a **lock-R** on *D*
        then
          upgrade lock on *D* to **lock-W**
        else
          grant *Ti* a **lock-W** on *D*
        write(*D*)
    end;
- All locks are released after commit or abort

Deadlock Handling

- Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

- ***Deadlock prevention*** protocols ensure that the **system will *never* enter into a deadlock state.**

- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration or conservative two phase locking).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol)

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

- **wait-die** scheme — non-preemptive
  - If TS(Ti)<TS(Tj) then Ti is allowed to wait otherwise abort Ti and restart it later with the same timestamp.
  - Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item

- **wound-wait** scheme — preemptive
  - If TS(Ti)<TS(Tj) then abort Tj and restart it later with the same timestamp otherwise Ti is allowed to wait
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.

Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

- Timeout-Based Schemes:
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

- Deadlock Detection where we periodically check to see if the system is in a state of deadlock. This can happen if the transaction weight is less and so they repeatedly lock the item.

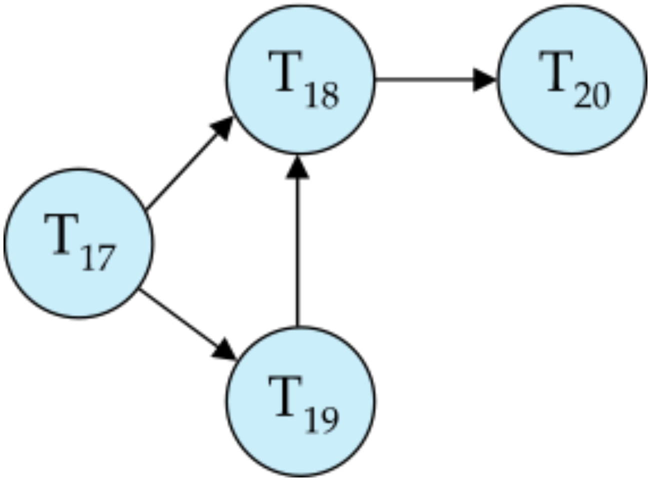
- Deadlocks can be described as **a wait-for graph**, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.

- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

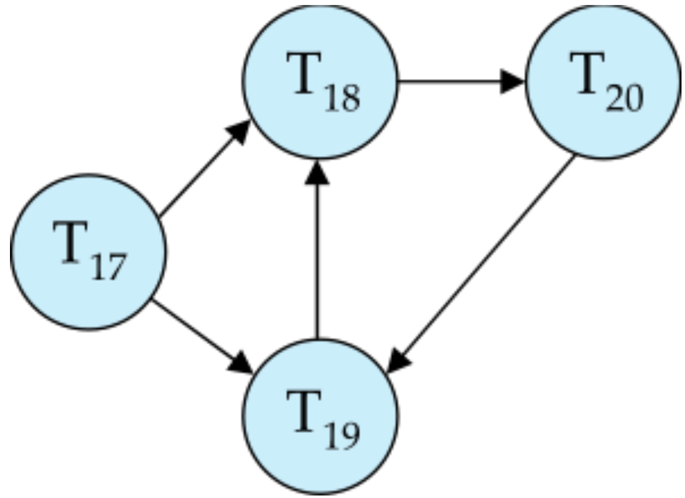
- When T_i requests a data item currently being held by T_j, then the edge T_i T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.

- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS(*Ti*), a new transaction *Tj* is assigned time-stamp TS(*Tj*) such that TS(*Ti*) <TS(*Tj*).

- The protocol manages concurrent execution such that the time-stamps determine the serializability order.

- In order to assure such behavior, the protocol maintains for each data '*Q*' two timestamp values:
  - **W-timestamp**(*Q*) is the largest time-stamp (Youngest Transaction) of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

- Suppose a transaction Ti issues a **read**(Q)
1. If TS(Ti) □ **W**-timestamp(Q), then Ti needs to read a value of Q that was already overwritten.
   - Hence, the **read** operation is rejected, and Ti is rolled back.
2. If TS(Ti)□ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(Ti)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction *Ti* issues **write**(*Q*).

1. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced.
   - Hence, the **write** operation is rejected, and *Ti* is rolled back.
2. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*.
   - Hence, this **write** operation is rejected, and *Ti* is rolled back.
3. Otherwise, the **write** operation is executed, and W-timestamp(*Q*) is set to TS(*Ti*).

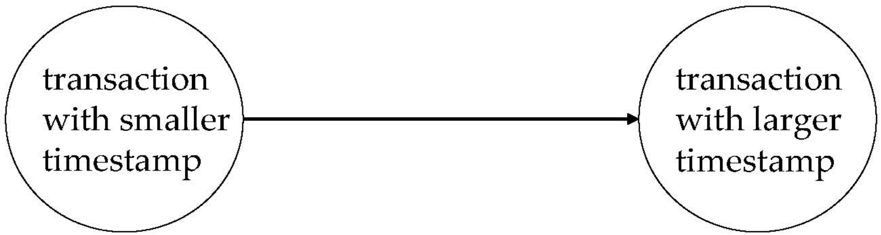
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5. Apply Timestamp based protocol.

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol **ensures freedom from deadlock** as no transaction ever waits.

- But the schedule may **not be cascade-free**, and may not even be **recoverable:** Due to absence of Locking mechanism

Validation-Based Protocol

- Execution of transaction *Ti* is done in three phases.
  1. **Read and execution phase**: Transaction *Ti* writes only to temporary local variables
  2. **Validation phase**: Transaction *Ti* performs a ``validation test'' to determine if local variables can be written without violating serializability.
  3. **Write phase**: If *Ti* is validated, the updates are applied to the database; otherwise, Ti is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol (Cont.)

- Each transaction Ti has 3 timestamps
  - Start(Ti) : the time when Ti started its execution
  - Validation(Ti): the time when Ti entered its validation phase
  - Finish(Ti) : the time when Ti finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS(Ti) is given the value of Validation(Ti).

Validation Test for Transaction *Tj*

- To handle Read and Write phases clash
- If for all *Ti* with TS (*Ti*) < TS (*Tj*) either one of the following condition holds:
  1. **finish**(*Ti*) < **start**(*Tj*) Tj's validation test.
  2. **finish**(*Ti*) < **validation**(*Tj*) **and** the set of data items written by *Ti* does not intersect with the set of data items read by *Tj*.

then validation succeeds and *Tj* can be committed.

Otherwise, validation fails and *Tj* is aborted.

- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of *Ti* do not affect reads of *Tj* since they occur after *Ti* has finished its reads.
  - the writes of *Ti* do not affect reads of *Tj* since *Tj* does not read any item written by *Ti.*

Validation protocol

Cascade-less : Since Validation test make sure that Ti reads only the committed values

No deadlock : Since make use of Timestamp

Suffer from Starvation

When validation test fails.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B \quad 50$
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

End of Chapter

- Thanks to Alan Fekete and Sudhir Jorwekar for Snapshot Isolation examples

Figure 15.01

	S	X
S	true	false
X	false	false

Figure 15.04

T_1	T_2	concurrency-control manager
lock-x (B)		
		grant-x (B, T_1)
read (B)		
$B := B - 50$		
write (B)		
unlock (B)		
	lock-s (A)	
		grant-s (A, T_2)
	read (A)	
	unlock (A)	
	lock-s (B)	
		grant-s (B, T_2)
	read (B)	
	unlock (B)	
	display ($A + B$)	
lock-x (A)		
		grant-x (A, T_2)
read (A)		
$A := A + 50$		
write (A)		
unlock (A)		

Figure 15.07

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

Figure 15.08

T_5	T_6	T_7
lock-x (A)		
read (A)		
lock-s (B)		
read (B)		
write (A)		
unlock (A)		
	lock-x (A)	
	read (A)	
	write (A)	
	unlock (A)	
		lock-s (A)
		read (A)

Figure 15.09

T_8	T_9
lock-s (a_1)	
	lock-s (a_1)
lock-s (a_2)	
	lock-s (a_2)
lock-s (a_3)	
lock-s (a_4)	
	unlock-s (a_3)
	unlock-s (a_4)
lock-s (a_n)	
upgrade (a_1)	

Figure 15.10

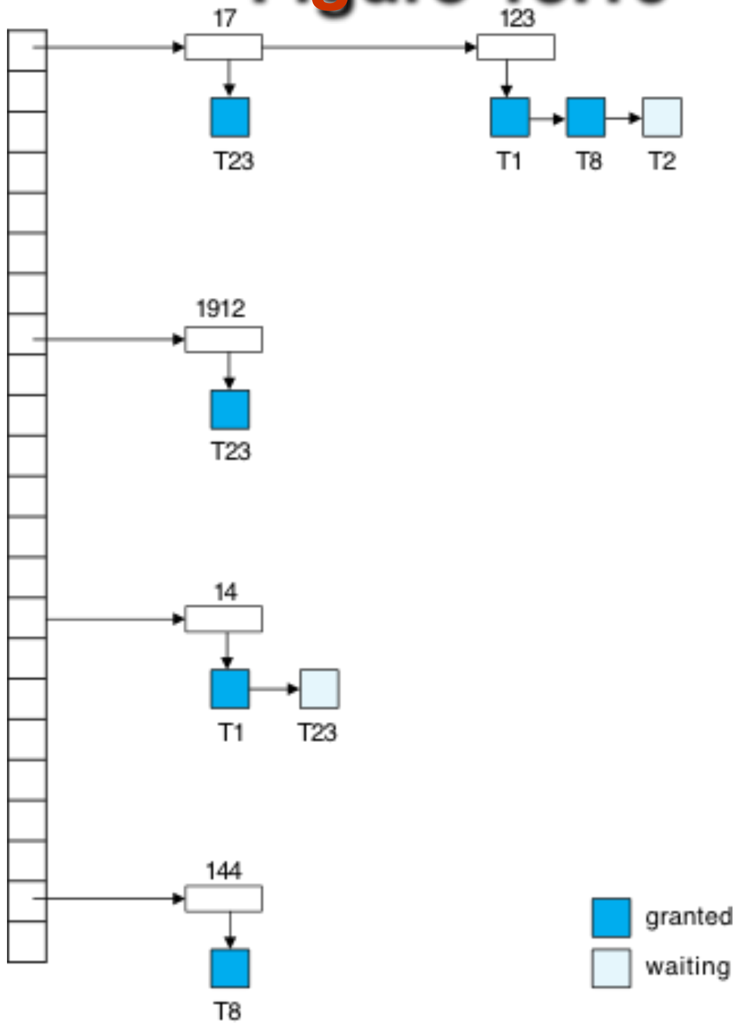


Figure 15.11

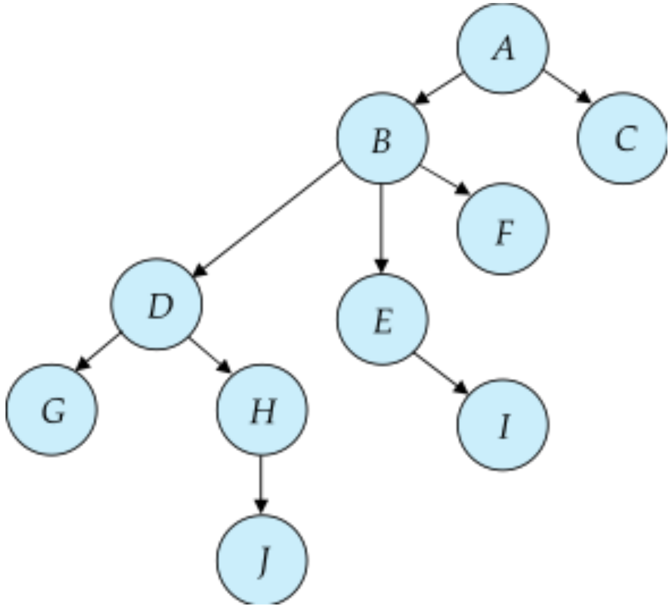


Figure 15.12

T_{10}	T_{11}	T_{12}	T_{13}
lock-x (B)			
	lock-x (D)		
	lock-x (H)		
	unlock (D)		
lock-x (E)			
lock-x (D)			
unlock (B)			
unlock (E)			
		lock-x (B)	
		lock-x (E)	
	unlock (H)		
lock-x (G)			
unlock (D)			
			lock-x (D)
			lock-x (H)
			unlock (D)
			unlock (H)
		unlock (E)	
		unlock (B)	
unlock (G)			

Figure 15.13

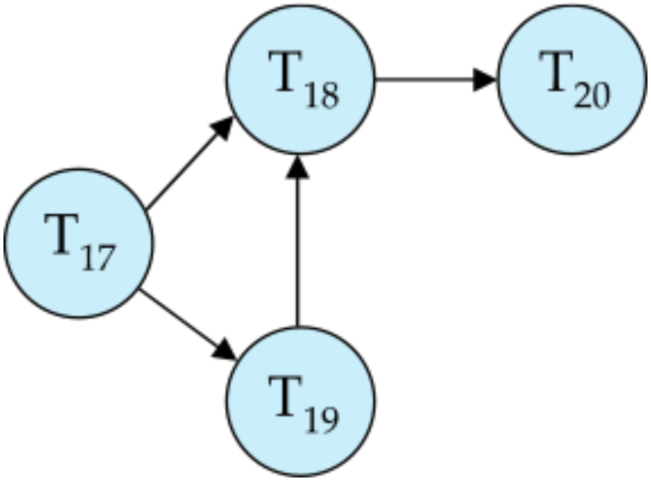


Figure 15.14

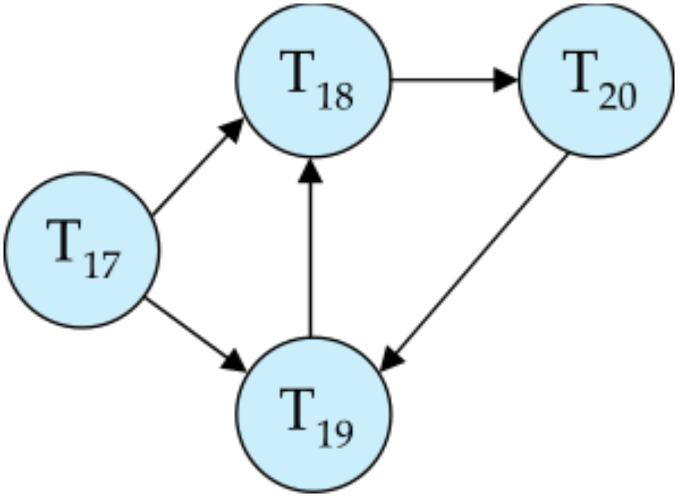


Figure 15.15

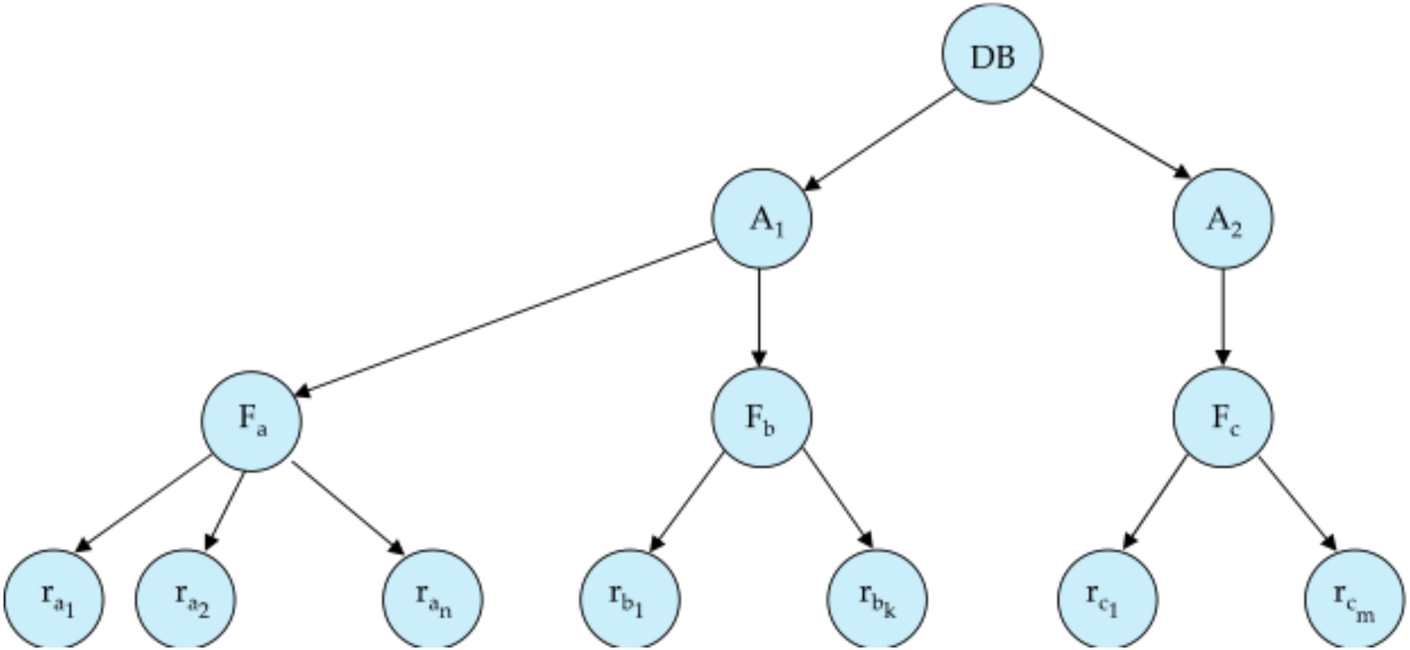


Figure 15.16

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 15.17

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B - 50$
	write (B)
read (A)	
	read (A)
display ($A + B$)	
	$A := A + 50$
	write (A)
	display ($A + B$)

Figure 15.18

T_{27}	T_{28}
read (Q)	
	write (Q)
write (Q)	

Figure 15.19

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B \quad 50$
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

Figure 15.20

T_{32}	T_{33}
lock-s (Q)	
read (Q)	
unlock (Q)	
	lock-x (Q)
	read (Q)
	write (Q)
	unlock (Q)
lock-s (Q)	
read (Q)	
unlock (Q)	

Figure 15.21

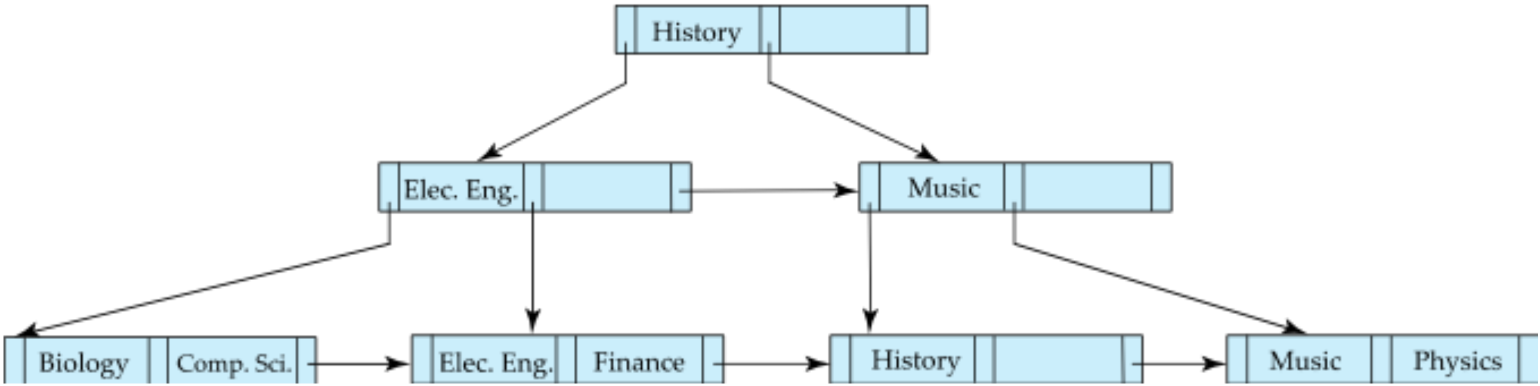

History
Elec. Eng.
Music
Biology
Comp. Sci.
Elec. Eng.
Finance
History
Music
Physics

Figure 15.22

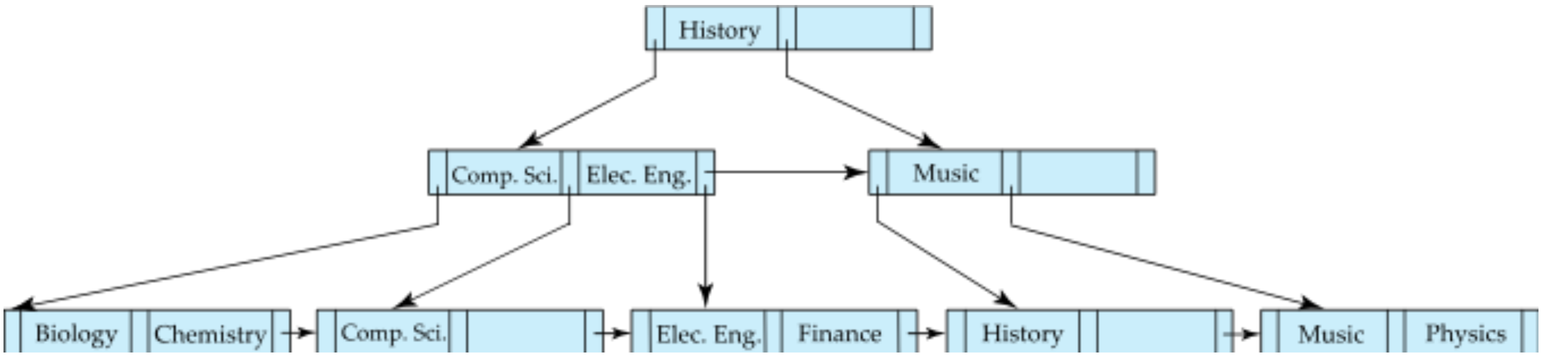

History
Comp. Sci.
Elec. Eng.
Music
Biology
Chemistry
Comp. Sci.
Elec. Eng.
Finance
History
Music
Physics

Figure 15.23

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

Figure in-15.1

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)