Chapter 15 : Concurrency Control

Database System Concepts, 6th Ed.

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

관전 Point ➔ Update conflict를 발생시키는 mechanism의 이해

Lock-Based Protocols [1/3]

- A lock is a mechanism to control concurrent access to a data item
- 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 the concurrency-control manager
- Transaction can proceed only after a lock request is granted

Lock-Based Protocols [2/3]

- **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 lock 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

** Compatible 하지 않는 Lock은 현재의 Lock이 풀릴 때 까지 기다려야 함

Lock-Based Protocols [3/3]

- Example of a transaction performing locking:

T_2: **lock-S**(*A*);
read (*A*);
unlock(*A*);
lock-S(*B*);
read (*B*);
unlock(*B*);
display(*A*+*B*)

Current Value ➔ A = 100, B = 50

Update ➔ A = 80, B = 80

A = 100, B = 80

- 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
  - X = read(A), Y = read(B), display(X, Y) 로 보면 더 clear
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks
  - The locking protocols restrict the set of possible schedules
  - The locking protocol must ensure serializability

B의 계좌에서 A의 계좌로
50$를 계좌이채

Fig 16.4 Schedule for Transactions: non-serializable schedule

Sample Transactions with Locks

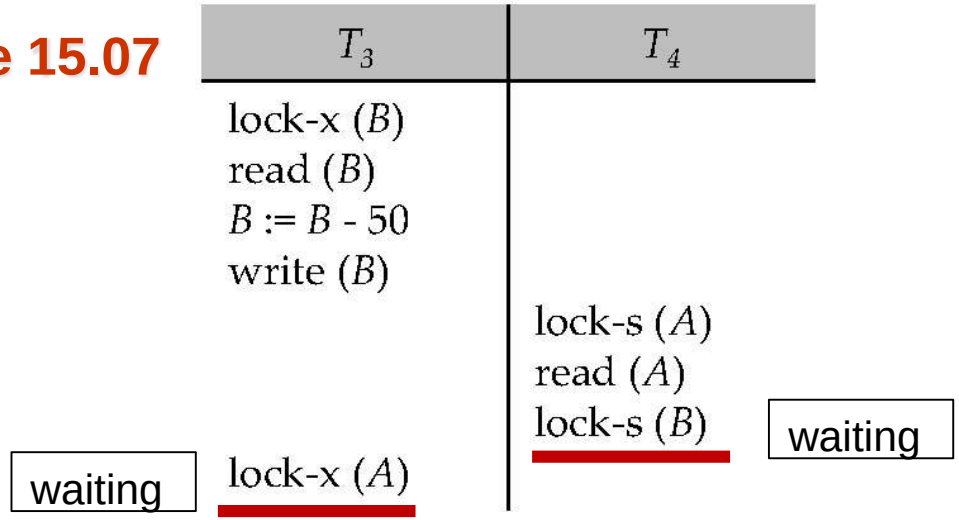
- T1: lock-X(B) T2: lock-S(A)
  read(B) read(A)
  B = B -50; unlock(A)
  write(B); lock-S(B)
  unlock(B); read(B);
  lock-X(A); unlock(B);
  read(A); display(A+B);
  A = A + 50;
  write(A);
  unlock(A);

T1	T2	concurrency-control manager
lock-X(B)		
		grant-X(B, T1)
Read(B)		
B = B - 50;		
write(B);		
unlock(B);		
	lock-S(A)	
		grant-S(A, T2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T2)
	read(B);	
	unlock(B);	
	display(A+B);	
lock-X(A);		
		grant-X(A, T2)
read(A);		
A = A + 50;		
write(A);		
unlock(A);		

Pitfalls of Lock-Based Protocols [1/2]

- Consider the partial schedule

Figure 15.07



- Neither T_3 nor T_4 can make progress
  - executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*

- Such a situation is called a **deadlock**
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released

Pitfalls of Lock-Based Protocols [2/2]

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

- **Starvation**
  - If concurrency control manager is badly designed and the same transaction is kept waiting during the deadlock resolution process
  - Example: A transaction T 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 T is repeatedly rolled back due to deadlocks
  - Concurrency control manager should be designed to prevent starvation

The Two-Phase Locking Protocol [1/4]

- Eswaran, Gray, Lorie, and Traiger, “The Notion of Consistency and Predicate Locks in a Database System” CACM 1976
- The 2PL protocol 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 2PL 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)
- The 2PL protocol *does not* ensure freedom from deadlocks

Serializable Schedule Examples under 2PL

T1	T2
lock-X(A)	
read(A)	
A←A+100	
write(A)	
lock-X(B)	
unlock(A)	
	lock-X(A)
	read(A)
	A←A*2
	write(A)
read(B)	
B←B+100	
write(B)	
unlock(B)	
	lock-X(B)
	unlock(A)
	read(B)
	B←B*2
	write(B)
	unlock(B)

= {T1; T2}

T1	T2
lock-S(X)	
A1←read(X)	
A1←A1-K	
lock-X(X)	
	lock-S(X)
X←A1	
write(X)	
lock-S(Y)	
A2←read(Y)	
A2←A2+K	
lock-X(Y)	
Y←A2	
write(Y)	
unlock(X)	
	A1←read(X)
	A1←A1*0.01
	lock-X(X)
	X←A1
	write(X)
	lock-S(Y)
unlock(Y)	
	A2←read(Y)
	A2←A2*0.01
	lock-X(Y)
	Y←A2
	write(Y)
	unlock(X)
	unlock(Y)

= {T1; T2}

The Two-Phase Locking Protocol [2/4]

- Cascading roll-back is possible under two-phase locking
  - Early lock release (than commit point) may cause cascading roll-back
  - T5가 commit하기전에 unlock(A)를 했으므로 T5가 roll-back한다면 T6와 T7은 cascading roll-back해야 함

Fig 15.8 Partial Schedule Under 2PL Showing the Possibility of Cascading Rollback

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)	
⇨ Abort or Commit		lock-S(A)
		read(A)

T5의 unlock(A)를 ⇨ point까지 유지했다면 T6의 lock-X(A)도 ⇨ point 뒤로 갈것이고 cascading roll-back이 필요없다

The Two-Phase Locking Protocol [3/4]

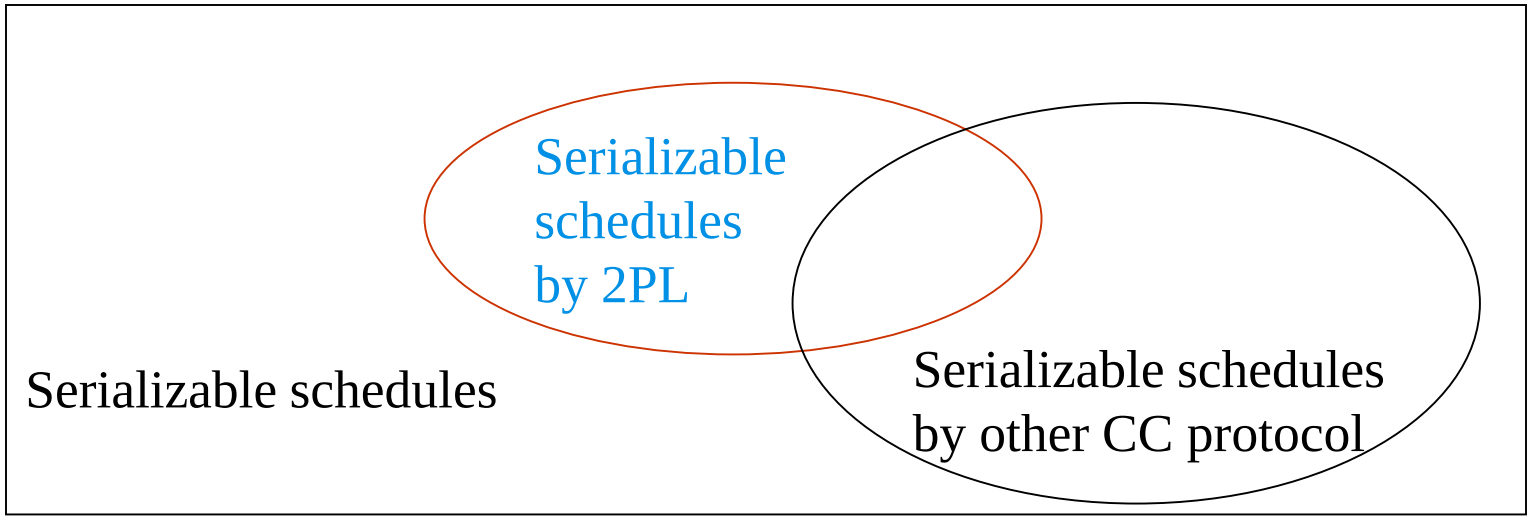
- **The regular 2PL은 unlock을 commit 시점보다 일찍하는것을 허락**
- **Strict 2PL**
  - Here a transaction must hold all its exclusive locks till it commits/aborts
  - No cascading rollback
- **Rigorous 2PL** is even stricter:
  - Here *all* locks(shared and exclusive) are held till commit/abort
  - No cascading rollback (of course)
  - In this protocol transactions can be serialized in the order in which they commit

- Comparing with the regular 2PL
  - Concurrency is limited
  - Deadlock can still happen

The Two-Phase Locking Protocol [4/4]

- There can be conflict serializable schedules that cannot be obtained if 2PL is used
- However, in the absence of extra information (e.g., ordering of access to data), 2PL is needed for conflict serializability in the following sense:

  Given a transaction T_i that does not follow 2PL, we can find a transaction T_j that uses 2PL, and a schedule for T_i and T_j that is not conflict serializable



2PL with Lock Conversions

- The original lock mode with (lock-X, lock-S)
  - assign lock-X on a data D when D is both read and written
- 2PL with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)

- This protocol assures serializability
- The refined 2PL with lock conversion gets more concurrency than the original 2PL

Fig 16.9 Incomplete Schedule With a Lock Conversion

** Unless lock conversion, the first step "lock-S (a1)" in T8 should be "lock-X (a1)", then T8 and T9 in the following schedule cannot be run concurrently in the original 2PL

** However, T8 and T9 can run concurrently in the refined 2PL owing to the lock conversion

T8	T9
read (a1)	read (a1)
read (a2)	read (a2)
…..	a3 = a2 – a1
read (an)	
a1 = a1 + a2 … an	
write(a1)	

The Original 2PL

T8	T9
lock-X (a1)	
lock –S (a2)	
…..	
lock-S (an)	
unlock (a1)	
	lock-S (a1)
	lock-S (a2)

2PL with Lock Conversion

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 (a_1)
	unlock (a_2)
lock-S (a_n)	
upgrade (a_1)	

- Strict 2PL (with Lock conversions) & Rigorous 2PL (with Lock conversions) are used extensively in commercial DBMSs

Automatic Acquisition of Locks [1/2]

- A transaction T_i issues the standard read/write instruction, without explicit locking calls
  - read와 write 연산 내에서 자동적으로 이루어짐

- The operation **read**(*D*) is processed as:

  if T_i has a lock on *D*

  then

  read(*D*)

  else begin

  if necessary, wait until no other transaction has a **lock-X** on *D;*

  grant T_i a **lock-S** on *D*;

  read(*D*)

  end

Machine-Level Lock Implementation

- The **test-and-set** instruction is an (CPU) instruction used to write to a memory location and return its old value as a single atomic operation
  - Typically, the value 1 is written to the memory location
  - If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished

- Lock can be built using an atomic test_and_set instruction

```
function Lock(boolean *lock)  {
      while (test_and_set(lock) == 1) ;
 }
```

```
function UnLock(boolean *lock)  {
      lock = 0 ;
 }
```

  - The process calling Lock() obtains the lock if the old value was 0 (a Boolean value); otherwise the calling process waits (spins)
  - Obtaining a lock means writing 1 (a Boolean value) to the variable
  - The process calling UnLock() write 0 (a Boolean value) to the variable

- The **test-and-set** instruction is also used for OS Mutual Exclusion, OS Semaphore

Automatic Acquisition of Locks [2/2]

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

```
if Ti has a  lock-X on D
 then   write(D)
 else begin
     if necessary, wait until no other transaction has any lock on D,
     if Ti has a lock-S on D      // 만약 lock-S를 가졌다면 lock-conversion
         then   upgrade lock on D  to lock-X
         else   grant Ti a lock-X on D
         write(D)
  end;
```

- All locks are released after commit or abort

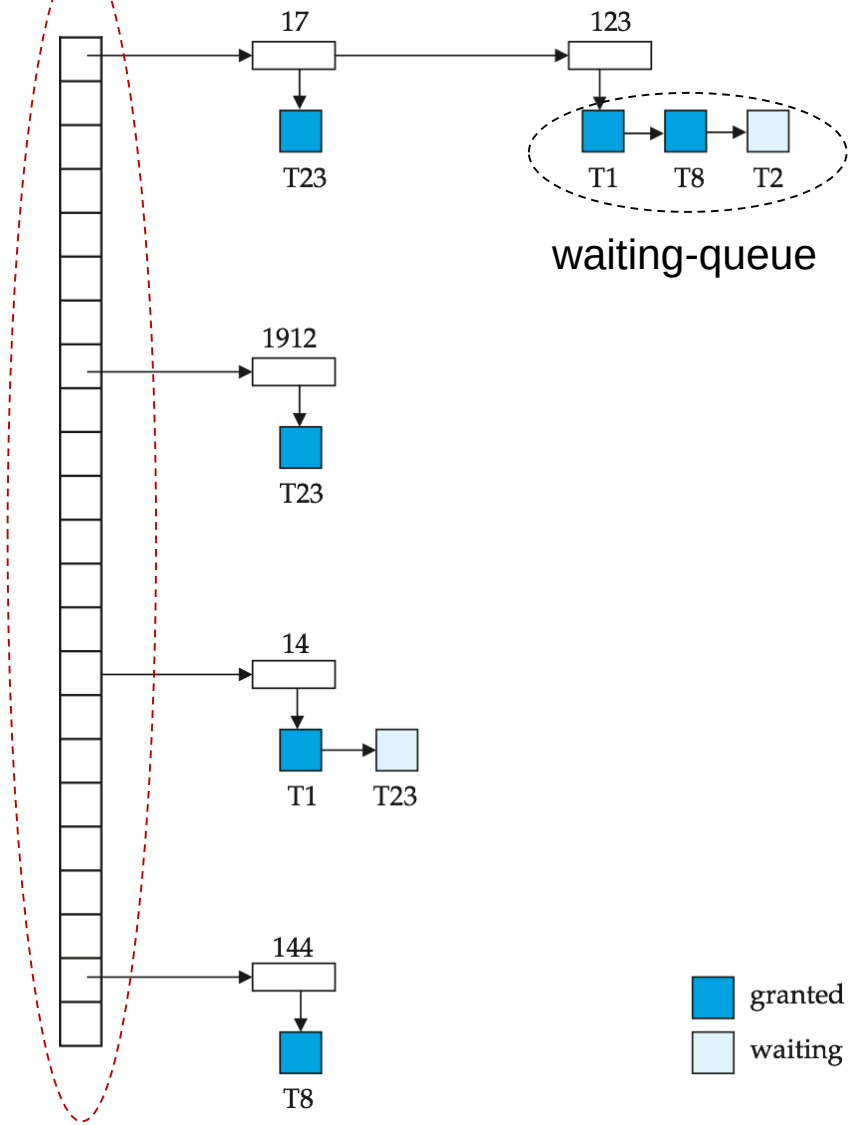
Implementation of Lock Manager

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
  - replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
  - The requesting transaction waits until its request is answered

- The lock manager maintains a data-structure called a **lock table**
  - To record granted locks and pending requests
  - The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked
    - **Hashing with overflow chaining**
    - **Lock request on item I ➔ Hashing (I) & enqueue & wait**
    - **Lock granted**
    - **Unlock request ➔ dequeue**

Lock Table Structure

Figure 15.10



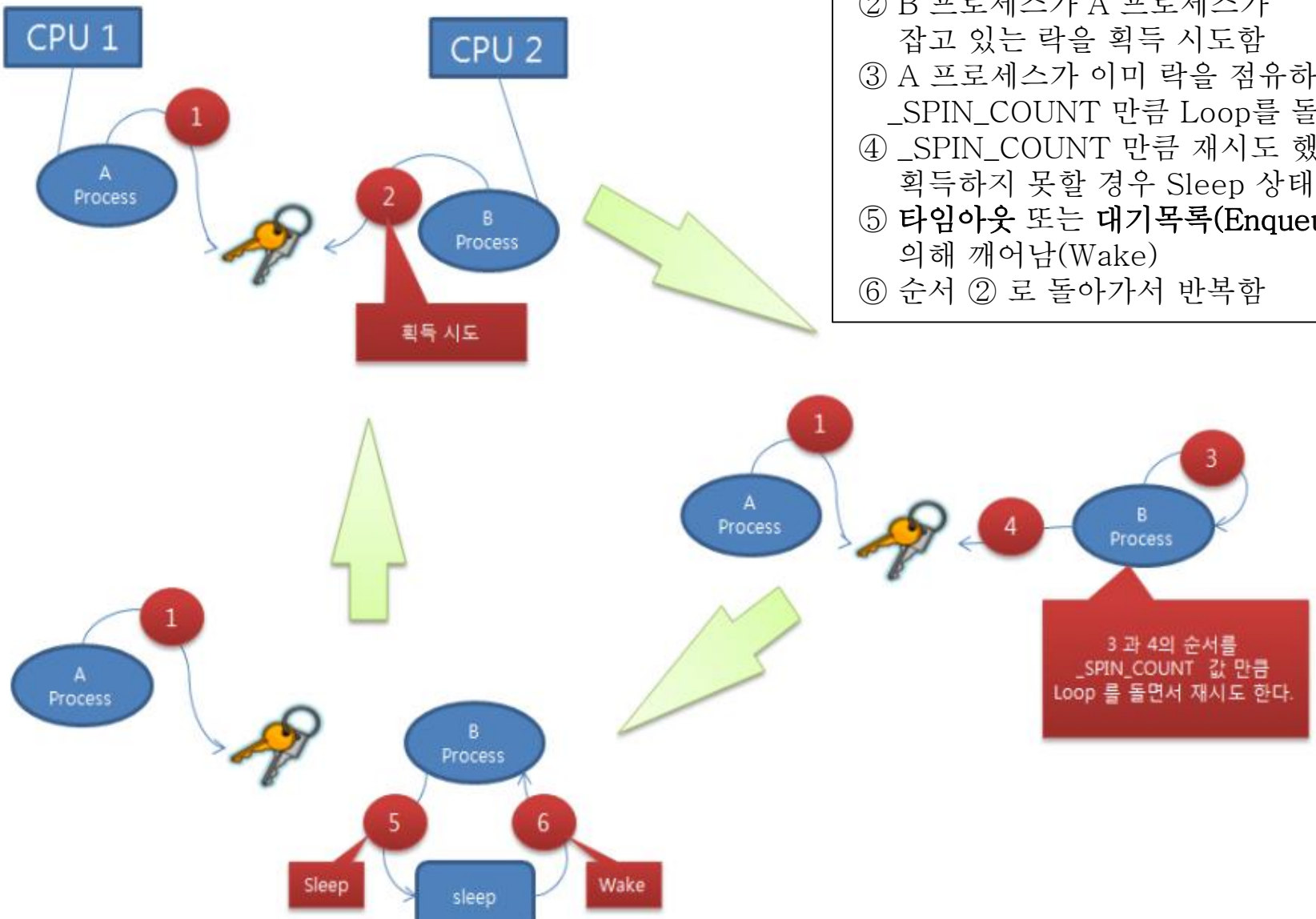
- White Box is a data item
- Blue rectangles indicate granted locks, light shader rectangles indicate waiting requests
- Lock table also records the type of lock granted or requested (nor shown here)
- Lock manager processes
  - New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
  - Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
  - If transaction aborts, all waiting or granted requests of the transaction are deleted
  - Lock manager may keep a list of locks held by each transaction, to implement this efficiently

Mechanism of Acquiring Lock [1/2]

참고자료

Willing-to-wait 방식: 원하는 락을 획득실패 했을때 락을 획득할 수 있을 때까지 대기

① A 프로세스가 락을 잡고 있음
② B 프로세스가 A 프로세스가 잡고 있는 락을 획득 시도함
③ A 프로세스가 이미 락을 점유하고 있으므로 _SPIN_COUNT 만큼 Loop를 돌면서 재시도함
④ _SPIN_COUNT 만큼 재시도 했음에도 획득하지 못할 경우 Sleep 상태가 됨
⑤ **타임아웃** 또는 **대기목록(Enqueue)**에 의해 깨어남(Wake)
⑥ 순서 ② 로 돌아가서 반복함



Mechanism of Acquiring Lock [2/2]

참고자료

Lock을 기다릴때 spin하면서 test-and-set을 하다가 정해진 횟수가 넘으면 Sleep mode로 들어가거나 Enqueue를 한다

TimeOut

- Sleep 상태의 프로세스가 Wake 하는 방식 중, 타임아웃 방식은 Exponential backoff sleep 알고리즘으로 동작함.
- Sleep 상태가 될 때마다 타임아웃 시간을 두 배씩 증가시킴.
- 이러한 방식을 사용하는 이유는 과도한 SPIN을 방지하기 위함임.
- 또 무한정 대기목록에서 Sleep하는 것을 방지함

대기목록(Enqueue)

- 락 획득에 실패한 프로세스는 자신을 Enqueue 리소스의 대기목록(Waiter List)에 등록함
- 현재 락을 보유한 프로세스는 락의 사용이 끝나면 락을 해제하고 Enqueue 리소스의 대기목록에서 다음 프로세스를 깨워줌
- 깨어난 프로세스는 락이 Granted되어 dequeue되거나 여전히 락을 점유할 수 없을 경우 _SPIN_COUNT 만큼 Loop 시도 후 다시 Sleep 됨

Graph-Based Protocols [1/2]

- Silberschatz and Kedem, “Consistency in Hierarchical Database System”, JACM 1980
- Graph-based protocols are an alternative to 2PL
- Impose a partial ordering $\rightarrow$ on the set **D** = {d_1, d_2 ,..., d_h} of all data items
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*

- We use the term “The Tree Protocol”
  - The *tree-protocol* is a simple kind of graph protocol
  - The tree-protocol considers only exclusive locks
  - The tree-protocol guarantees freedom of deadlock

- The tree protocol is not practical, just meaningful from the view point of the CC theory

Tree Protocol

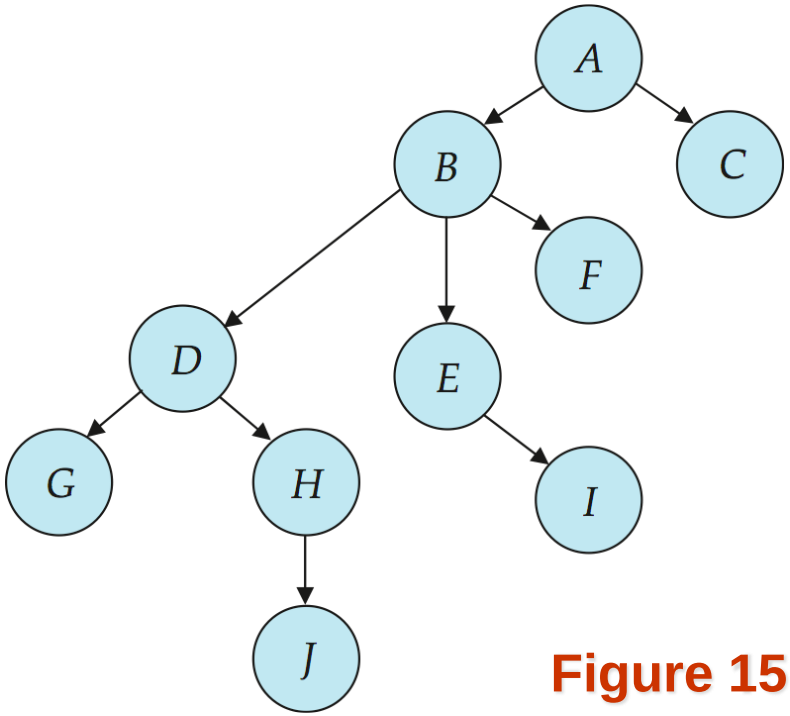
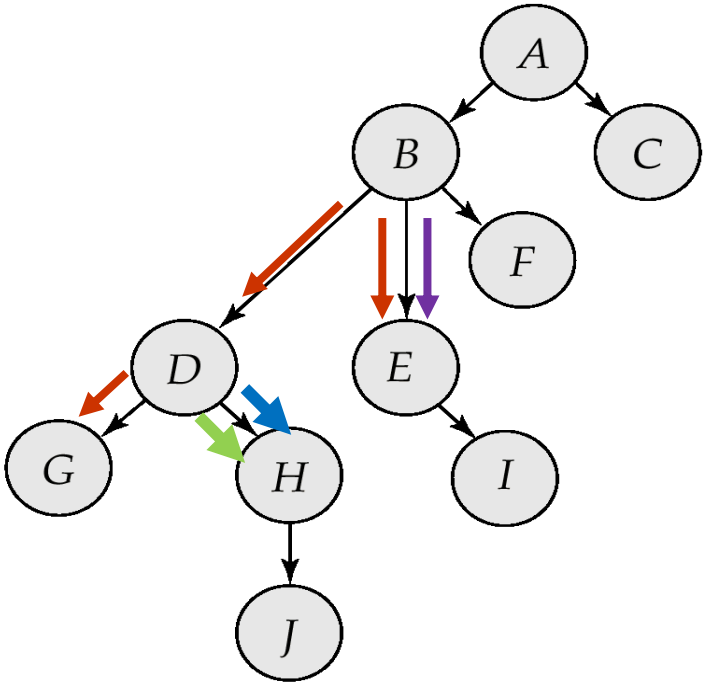


Figure 15.11

1. Only exclusive locks are allowed
2. The first lock by T_i may be on any data item if there is no lock on the data item
3. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i
4. Data items may be unlocked at any time

Fig 15.12 Serializable Schedule under the Tree Protocol

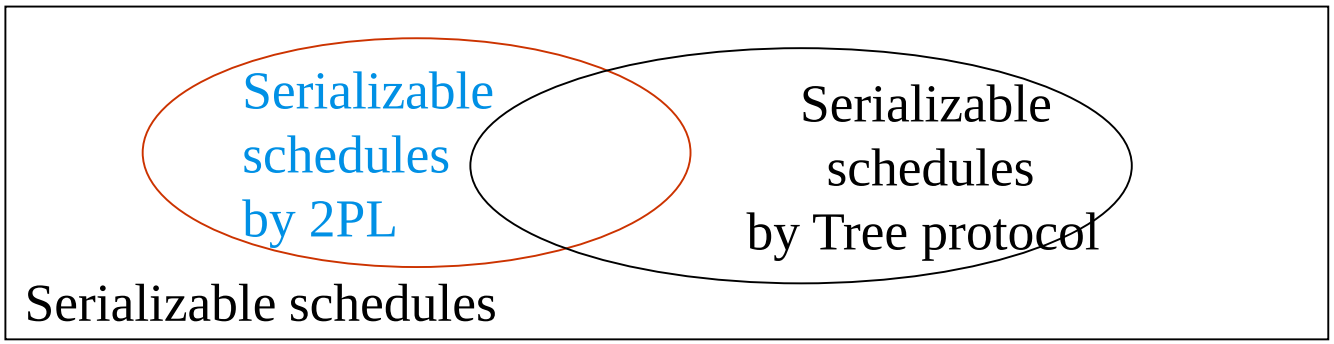
T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			



- 만약 T10 에서 G,B,D,E 순으로 락을 요청했다면, 트리 프로토콜에서는 허용하지 않는 스케줄이므로 거부됨
- B와 J만을 access하려는 Transaction은 D, H도 lock을 걸어야 한다

Graph-Based Protocols [2/2]

- The tree protocol ensures conflict serializability as well as freedom from deadlock
- Unlocking may occur earlier in the tree-locking protocol than in the 2PL protocol
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under 2PL are possible under tree protocol, and vice versa



Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

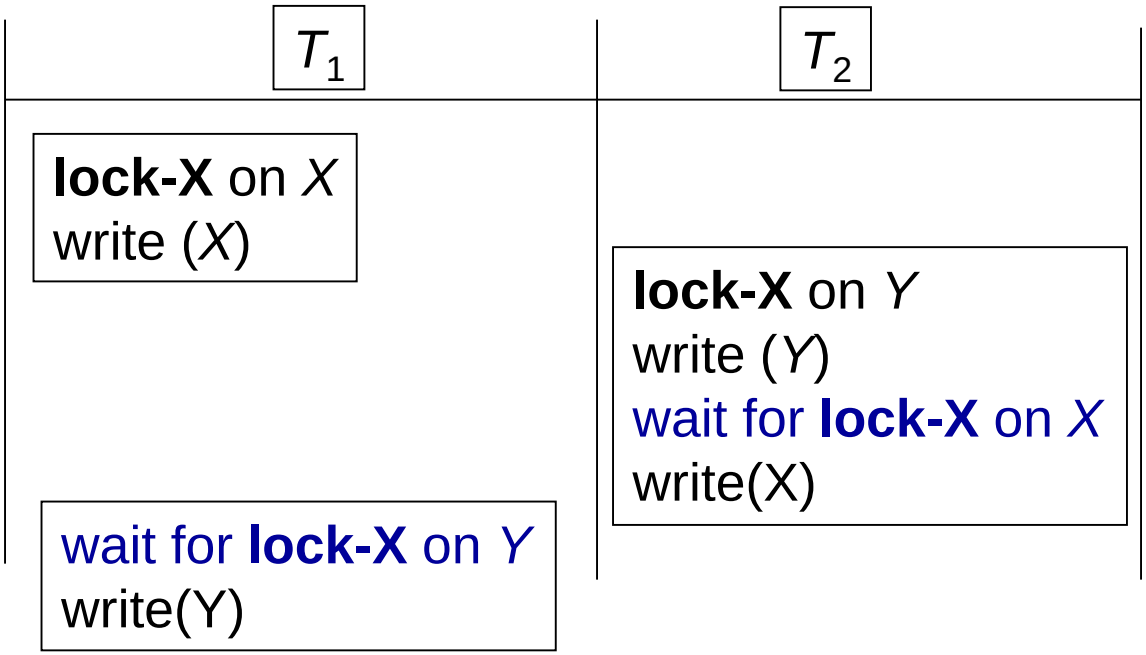
Deadlock Situation

- Consider the following two transactions:

  T_1: write (X) T_2: write(Y)

  write(Y) write(X)

- Schedule with deadlock



- Deadlock Handling
  - Deadlock prevention schemes
  - Deadlock detection schemes

Deadlock Prevention [1/2]

- 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
  - Require that each transaction locks all its data items before it begins execution (predeclaration)
  - Impose partial ordering of all data items
    - require that a transaction can lock data items only in the order specified by the partial order (e.g. graph-based protocol)
  - Timeout-Based Scheme:
    - a transaction waits for a lock only for a specified amount of time
      - After the wait time is out and the transaction is rolled back (No deadlock!)
    - simple to implement; but starvation is possible
    - Also difficult to determine good value of the timeout interval
  - Timestamping-based schemes (in the next slide)

Deadlock Prevention [2/2]

- **Timestamp-based deadlock prevention schemes**
- **wait-die scheme** — non-preemptive
  - 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
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it
  - Younger transactions may wait for older ones
  - May be fewer rollbacks than *wait-die* scheme

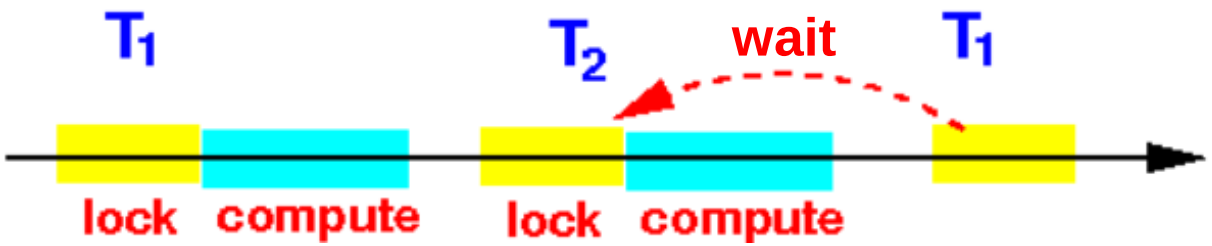
- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp
- Older transactions thus have precedence over newer ones, and starvation is hence avoided

TimeStamp based Deadlock Prevention :Wait-Die Scheme

참고자료

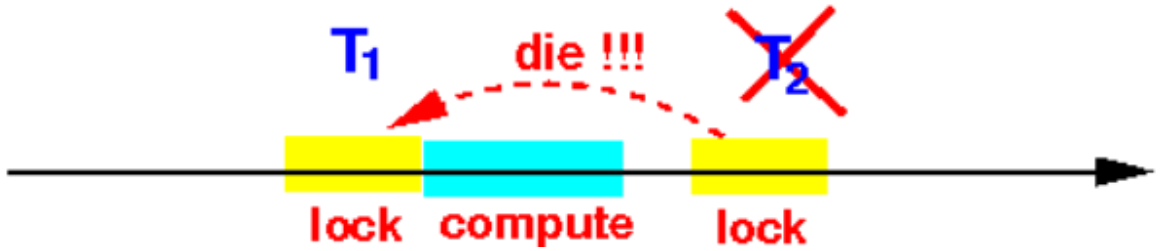
TS(T1) < TS(T2)
T1 : older transaction / T2 : newer transaction

Wait–die:



T1(older)이 T2(newer)가 점유한 락을 요청할 경우,
T1(older)은 기다린다. (Wait)

Wait–die:



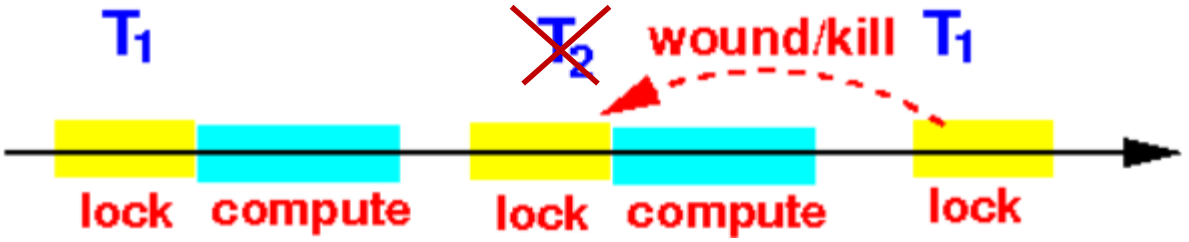
T2(newer)가 T1(older)이 점유한 락을 요청할 경우,
T2(newer)는 롤백한다. (Die)

TimeStamp based Deadlock Prevention :Wound-Die Scheme

참고자료

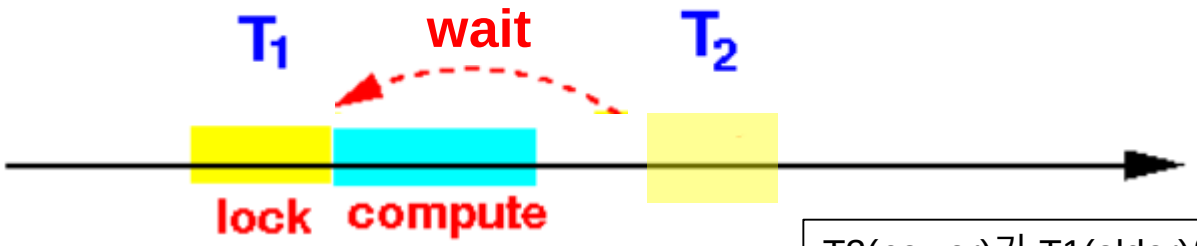
TS(T1) < TS(T2)
T1 : older transaction / T2 : newer transaction

Wound–wait:



T1(older)이 T2(newer)가 점유한 락을 요청할 경우,
T2(older)를 롤백시키고 T1(order)이 락을 획득한다. (Wound)

Wound–wait:



T2(newer)가 T1(older)이 점유한 락을 요청할 경우,
T2(newer)는 기다린다. (Wait)

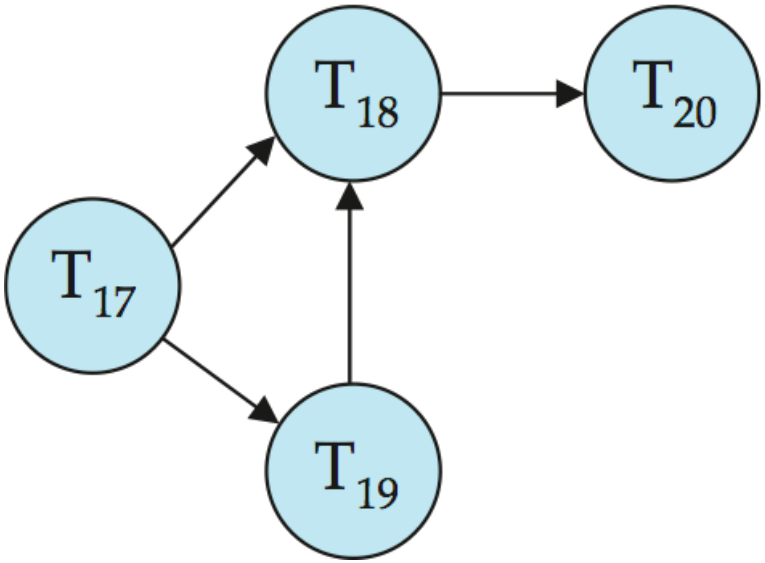
Deadlock Detection [1/2]

- 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 $T_i \rightarrow T_j$ 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

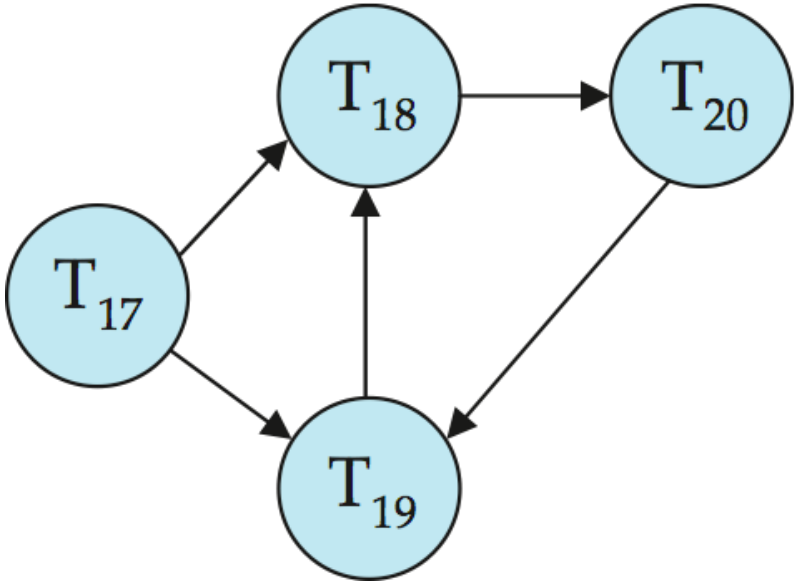
- The system is in a deadlock state if and only if the wait-for graph has a cycle
  - Must invoke a deadlock-detection algorithm periodically to look for cycles

Deadlock Detection [2/2]



Wait-for graph without a cycle

Figure 15.13



Wait-for graph with a cycle

Figure 15.14

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
    - **Partial rollback**: More effective to roll back transaction only as far as necessary to break deadlock
  - Starvation happens if same transaction is always chosen as victim
    - Need to keep the number of rollbacks in the cost factor to avoid starvation

- 현실에서 deadlock detection 방식은 사용되지 않음
- 트랜잭션들의 정보를 유지+ 주기적으로 detection 알고리즘을 가동 ➔ 오버헤드
- 데드락을 풀기 위해 어떤 트랜잭션을 롤백 할지에 대한 선택 비용 (진행이 적은 Transaction 선택)

Chapter 15: Concurrency Control

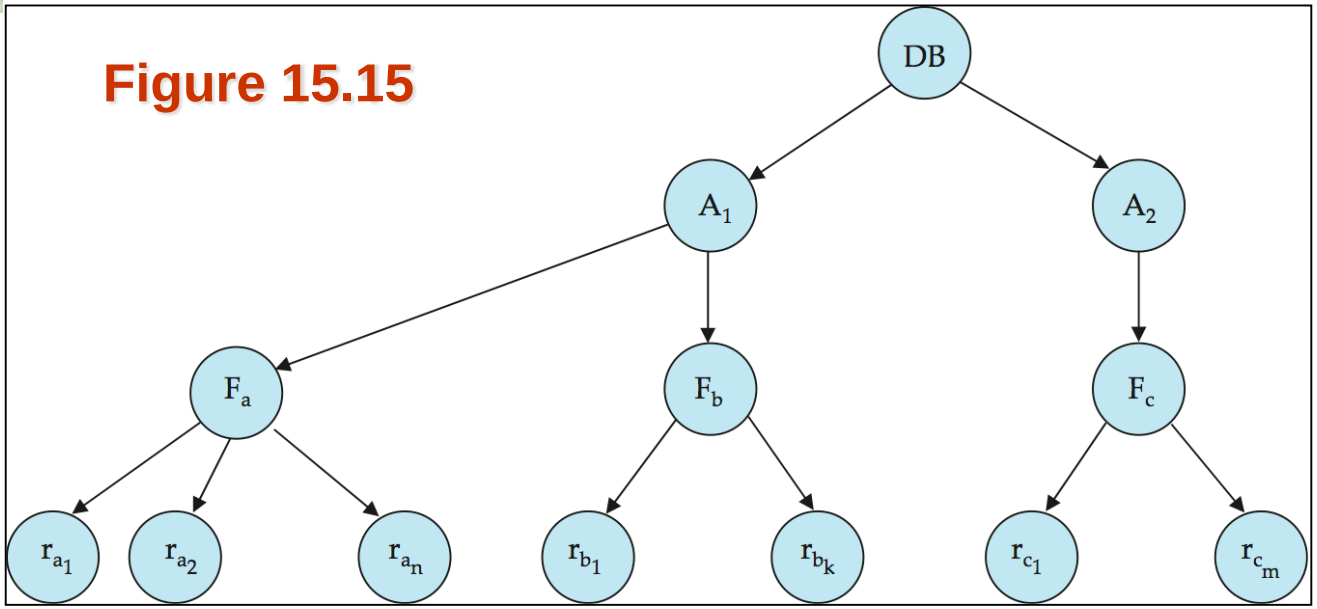
- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

Multiple Granularity Locking (MGL)

- Gray, Lorie, Putzolu, “Granularity of Locks and Degree of Consistency in a Shared Data Base”, VLDB 1975
  - Main Idea: Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree
  - but don't confuse with tree-locking protocol
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode
- **Granularity of locking** (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency

- The regular 2PL was on single data items
- Now the system (transaction manager) determines the granularity of locks after analyzing the concurrent transactions

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are:

- *Database* → *DB*
- *Area* → *A*
- *File* → *F*
- *record* → r

- TradeOff
  - 위쪽의 노드에 락을 걸면 locking overhead는 낮아지지만 concurrency가 나빠짐
  - 아래쪽의 노드에 락을 걸면 locking overhead는 높아지지만 concurrency는 좋아짐

Intention Lock Modes in MGL

- Higher level nodes include low-level nodes
  - Locks in higher level nodes are somehow related with locks in lower level nodes
- In addition to S and X lock modes, there are 3 additional lock modes with multiple granularity locking:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks
- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

Requesting

Holding

	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 15.16

	S	X
S	true	false
X	false	false

More potential concurrency!

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed
  2. The root of the tree must be locked first, and may be locked in any mode
  3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode
  4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase)
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order

- MGL Scheme is particularly useful in applications mixing
  - Short transactions that access only a few data items
  - Long transactions that produces reports from an entire file or set of files

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

Timestamp-Based Protocols [1/3]

- Reed, “Implementing Atomic Actions on Decentralized Data”, ACM TOCS 1983
  - Each transaction is issued a timestamp when it enters the system
    - 즉, concurrent transaction들의 수행순서를 timestamp로 control하는 방식
  - If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j)
  - 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 of any transaction that executed **write**(Q) successfully
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully

Transaction T
Timestamp(T)

DataItem Q
W-timestamp(Q)
R-timestamp(Q)

Timestamp-Based Protocols [2/3]

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

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

추상적으로 설명해보면,
아이템 Q를 read 할 때, 나보다 타임스탬프가 큰 놈이 먼저 Q를 write했다면 내 트랜잭션은 롤백
그 외에는 read 가 허용되고, 현재의 R-timestamp(Q)와 내 타임스탬프 중 큰 값을 R-timestamp(Q)로 저장
// 나보다 타임스탬프 큰 놈이 그 사이에 갱신했을 수도 있기 때문에 내 타임스탬프가 가장 크다는 보장 못함

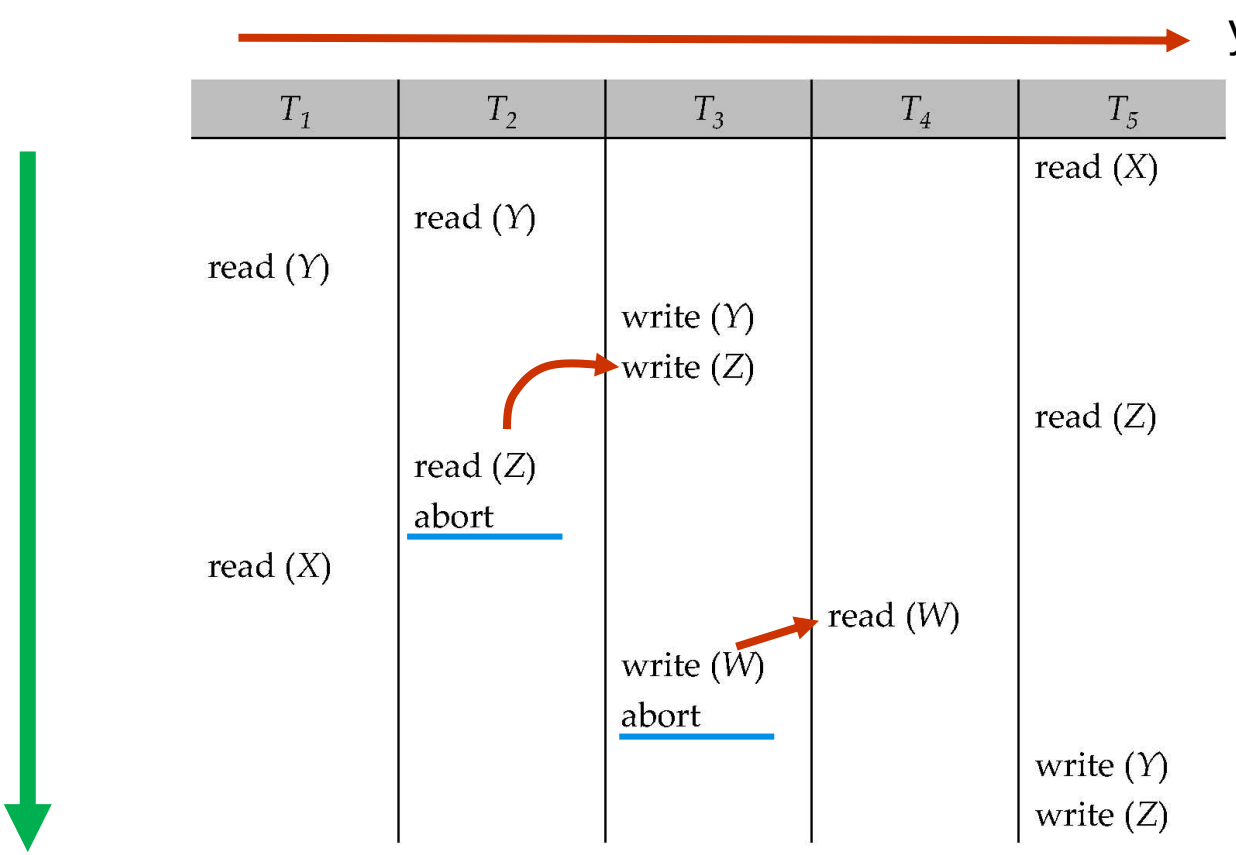
Timestamp-Based Protocols [3/3]

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

- 내가 아이템 Q를 write하려 할때 (1) 나 젊은놈이 먼저 read를 했다면 나는 롤백 또는 (2) 나보다 젊은놈이 먼저 write 했어도 나는 롤백: 만약 위의 두 roll-back상황 아니면 나의 write는 진행
- 나중에 step2는 합리적으로 개선되어서 roll-back을 줄이게 됨 (Thomas write rule)

Example of the Time-Stamp Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

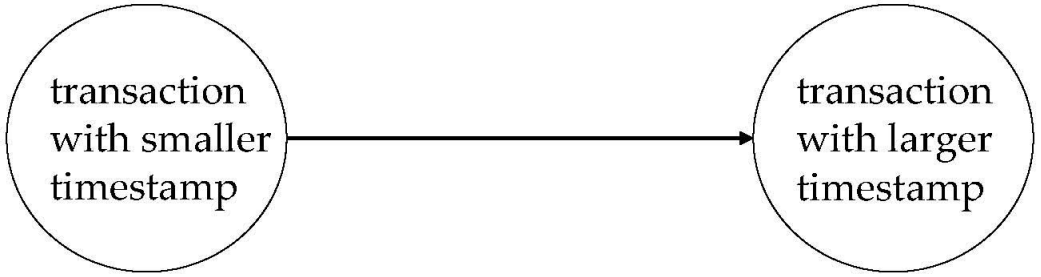


Concurrent Transaction들의 수행시간

나보다 젊은 Transaction이 (1) 먼저 write를 한 값을 내가 나중에 read하거나, (2) 먼저 read를 한 값을 내가 나중에 write했다면 나는 abort가 마땅하다

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
  - 왜냐면 timestamp protocol에 commit에 대한 조건이 없음
  - Write-read 종속성이 발생했을 때, read쪽의 transaction이 나중에 commit해야 된다는 recoverable에 관한 규정이 없음
  - 또 commit바로전에 unlock을 허락하는것 같은 cascadeless 관련 규정도 없음

Fig 15.17 Schedule 3:

Possible under both 2PL and the time stamping protocol

T_{14}	T_{15}
read (B)	
	read (B)
	$B := B - 50$
	write (B)
read (A)	
	read (A)
display $(A + B)$	
	$A := A + 50$
	write (A)
	display $(A + B)$

- Should wait in 2PL
- No waiting in Timestamp protocol

Recoverability and Cascade Freedom in Timestamping Protocol

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Timestamping Protocol with Thomas' Write Rule

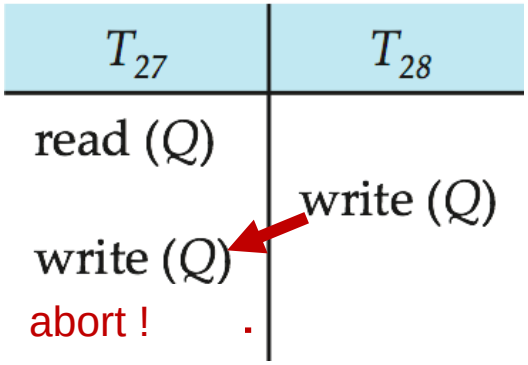
- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances
- Read part of Timestamping protocol is exactly same
- Modified Write part
  - When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}
    - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored
  - Otherwise this protocol is the same as the timestamp ordering protocol

- Thomas' Write Rule allows greater potential concurrency than the regular timestamping
  - Allows some view-serializable schedules that are not conflict-serializable

참고자료

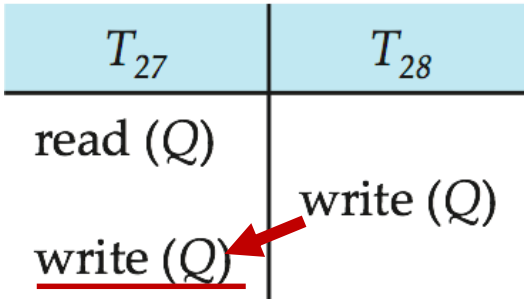
Thomas' Write Rule

기존의 Timestamp-Based Protocols



- T27이 write(Q)를 수행하면 기존의 Timestamp-Based Protocol 에서는 T27이 롤백 됨
- Thomas' Write Rule을 사용한 Timestamp-Based Protocol 에서는 T27의 write(Q)를 무시함으로써 T27이 롤백 되지 않음
- 즉 View Serializable Schedule을 허용할 수 있음

Timestamp-Based Protocols with Thomas' Write Rule



T27의 write(Q)를
무시함으로써 허용

나보다 젊은놈이 write를 했는데, 내가 지금 write을 해봐야 무슨소용이 있나!
(Q의 W-timestamp는 28인데, 27로 쓰는것은 소용이 없다)
그러니 abort하지 말고 그냥 진행해도 OK이다.

Remember?

View Serializability [1/2]

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S´* also transaction T_i must read the initial value of *Q*
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S´* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S´*

As can be seen, view equivalence is also based purely on **reads** and **writes** alone

Remember?

View Serializability [2/2]

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable
- Below is a schedule which is view-serializable but *not* conflict serializable

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Figure in-15.1

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**

- **위의 schedule은 conflict serializable하지 않지만 T6의 write(Q)에 의해서 실행의 결과가 T3, T4, T6한것과 같이 보이므로 view serializable하다고**

Other Notions of Serializability

Remember?

- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent, nor view equivalent to it
- Result equivalent

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

실행결과가 같다고 해서
view equivalent라고
100%이야기 할 수 없다

- Determining such equivalence requires analysis of operations other than read and write

Test for View Serializability

Remember?

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP-complete* problems.
  - Thus, the existence of an efficient algorithm is *extremely* unlikely.
- However, practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

Validation-Based Protocol [1/2]

- Kung and Robinson, “Optimistic Concurrency Control”, ACM TODS 1981
  - Execution of transaction T_i is done in 3 phases
    - **1. Read and execution phase**: Transaction T_i writes only to temporary local variables
    - **2. Validation phase**: Transaction T_i performs a ``validation test'' to determine if local variables can be written without violating serializability
    - **3. Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back
- The 3 phases of concurrently executing transactions can be interleaved, but each transaction must go through the 3 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

즉 트랜잭션이 성공적일 것이라고 예측하기 때문에, 일단 트랜잭션을 로컬에서 모두 수행시키는 단계와 DB에 반영하는 단계로 나눈 것임.
Timestamp protocol에서는 트랜잭션 수행 중에도 즉시 롤백 시키는 것과 대조적임 (pessimistic)

Validation-Based Protocol [2/2]

- Each transaction T_i has 3 timestamps:
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency
  - Thus TS(T_i) is given the value of Validation(T_i)
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back

Transaction T
Start(T)
Validation(T)
Finish(T)

일단은 진행하고 검사한 뒤 반영하므로 concurrency 좋음
read-only 트랜잭션이 많은 응용에 적합함 (conflict 적게 발생하는)
진행 중간중간에 검사해서 롤백하는 오버헤드 없음.
또 트랜잭션 단위의 타임스탬프만으로 프로토콜이 진행됨
(아이템 단위로 타임스탬프 저장하는 오버헤드 없음)

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition must hold:
  - **finish**(T_i) < **start**(T_j)
  - the set of data items written by T_i does not intersect with the set of data items read by T_j and T_i completes its write phase before T_j starts its validation phase (**start**(T_j) < **finish**(T_i) < **validation**(T_j) **)**

  then validation succeeds and T_j can be committed

  Otherwise, validation fails and T_j 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 T_j do not affect reads of T_i since they occur after T_i has finished its reads
    - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i

Schedule Produced by Validation Protocol

- Example of schedule produced using the validation protcol

Figure 15.19

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

T25, T26 순서대로 생성되었다면 T26의 validation phase에서 두 번째 조건을 만족하여 통과.
T25의 finish 시간이 T26의 start와 validation 타임스탬프 사이에 존재하며,
T25에서 갱신하려는 아이템과 T26에서 읽으려는 아이템이 겹치지 않음을 확인할 수 있음
(애당초 T25에서는 쓰려는 값이 없음),

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

Multiversion CC Schemes

- Reed, “Implementing Atomic Actions in Decentralized Data”, ACM TOCS, 1983
- Multiversion schemes keep old versions of data item to increase concurrency
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written
- Use timestamps to label versions
- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version
- **read**s never have to wait as an appropriate version is returned immediately

한 데이터 아이템에 대해 여러 버전을 유지해서,
트랜잭션이 적절한 시점의 버전을 읽고, 새로운 버전을 생성하는 것을 허용하는 기법
멀티버전 기법에서는 read 연산이 항상 보장되므로, read-only 트랜잭션이나 갱신연산이 적은 응용에서 사용하기 적합

Multiversion Timestamp Ordering [1/2]

- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i)
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k)

Transaction T
Timestamp(T)

DataItem Q1
W-timestamp(Q1)
R-timestamp(Q1)

→

DataItem Q2
W-timestamp(Q2)
R-timestamp(Q2)

→

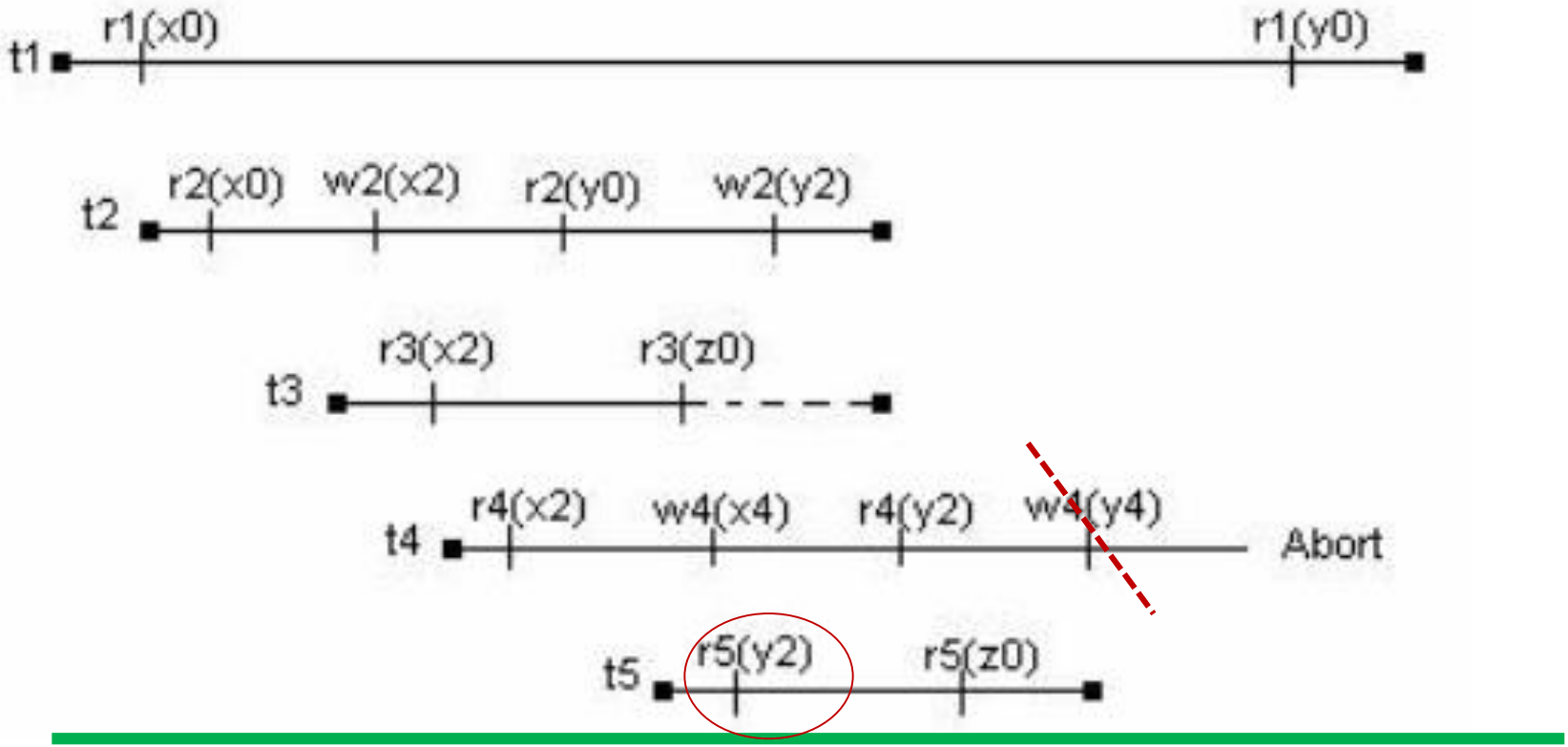
DataItem Q3
W-timestamp(Q3)
R-timestamp(Q3)

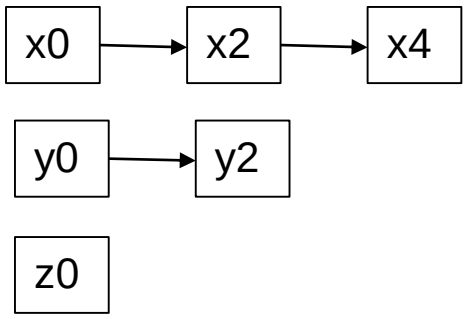
Multiversion Timestamp Ordering [2/2]

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation
- Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i)
  1. If transaction T_i issues a **read**(Q), then the value returned is the content of version Q_k
  2. If transaction T_i issues a **write**(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back
     2. Otherwise if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten, otherwise a new version of Q is created.
- Observe that
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i
- Protocol guarantees serializability

Multi Version Timestamp Ordering Example

참고자료





Multiversion Two-Phase Locking [1/2]

- Multiversion 2PL Protocol
  - MultiVersion Timestamp ordering + Rigorous 2PL
  - Differentiates between read-only transactions and update transactions
  - Ts-counter is a global time-stamp clock (logical counter)
- ***Update transactions*** acquire read and write locks, and hold all locks up to the end of the transaction (follow rigorous 2PL, thus no cascading roll-back)
  - Each successful **write** results in creating a new version of the data item written
  - Each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing
- ***Read-only transactions*** are assigned a timestamp by reading the current value of **ts-counter** before they start execution;
  - They follow the multiversion timestamp-ordering protocol for performing reads

Transaction T
Timestamp(T)

TS-Counter

DataItem Q5
Timestamp(Q5)

DataItem Q9
Timestamp(Q9)

Multiversion Two-Phase Locking [2/2]

- When an update transaction T_i wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version (< T_i)
- When an update transaction T_i wants to write an item,
  - it obtains X lock on;
  - it then creates a new version of the item and sets this version's timestamp to ∞
    - Commit전에는 다른 transaction이 읽을수 없도록
- When an update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter** + 1
  - T_i increments **ts-counter** by 1

- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i

- Only serializable schedules are produced

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g., if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

Active Transaction List: T9, T10, T12, ,,,,

DataItem Q5	**DataItem Q9**
Timestamp(Q5)	Timestamp(Q9)

Need to be
garbage collected

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

“Snapshot Isolation” Consistency Level

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update many rows
  - Poor performance results (= low concurrency)
  - Read-only transaction은 serializability를 안지켜도 되므로 weak consistency level들이 제시 (ANSI SQL consistency level)
  - “Snapshot isolation” consistency level by Berenson et al, SIGMOD 1995

- Solution 1: Multiversion + 2-phase locking
  - Give logical “snapshot” of database state to read-only transactions, read-write transactions use normal locking
  - Works well, but how does system know a transaction is read-only? ➔ overhead
- Solution 2: Simply give snapshot of database state to every transaction, updates alone use 2PL to guard against concurrent updates
  - Problem: May cause variety of anomalies such as lost update can result
- Solution 3: Add some more rules to Solution2
  - The snapshot isolation protocol (next slide)
  - Oracle, PostgreSQL, SQL Server 2005 are all taking this approach

Snapshot Isolation CC Scheme

- Fekete, Liarokapis, O'Neil, Shasha, "Making Snapshot Isolation Serializable", ACM TODS 2005
- 규약을 추가해서 기본적인 anomaly를 제거한 snapshot isolation CC scheme
- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write

DB = { x, y, z }

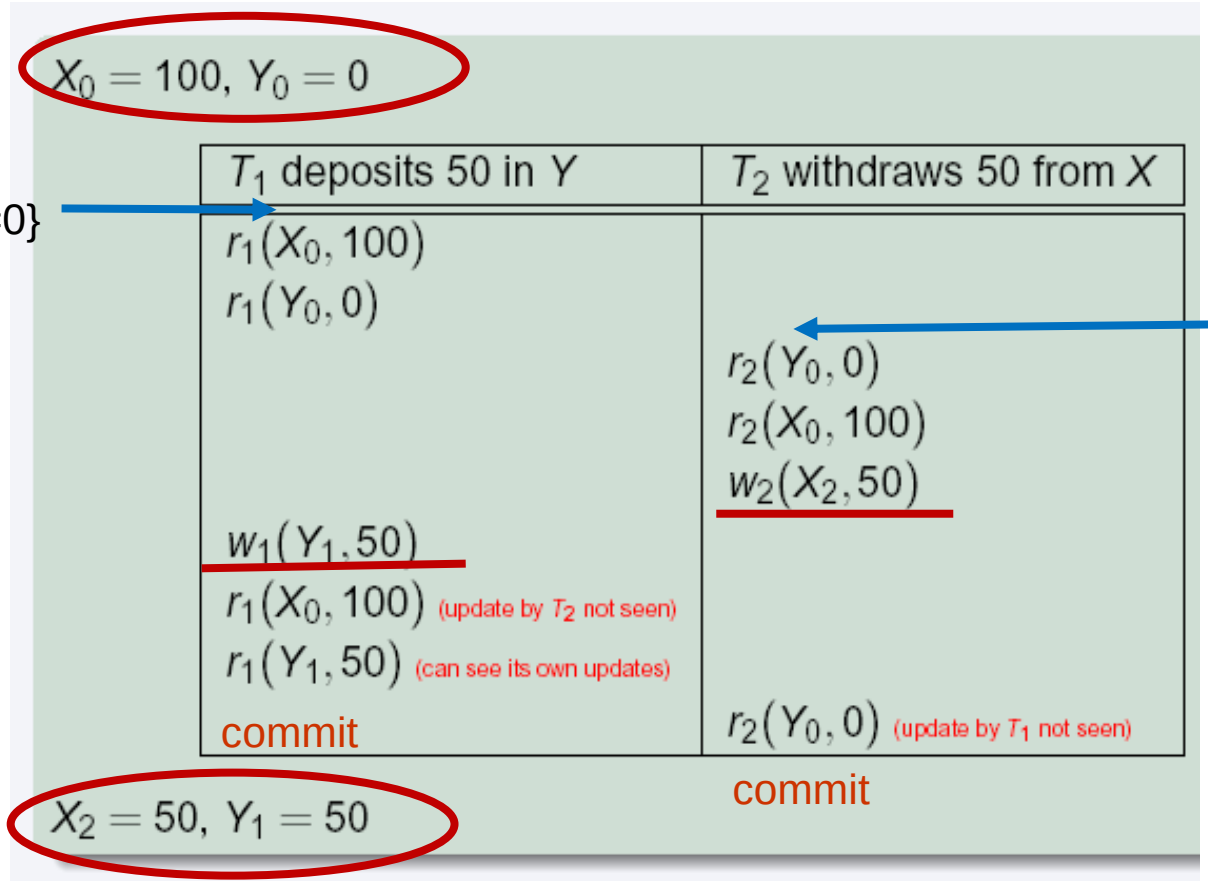
T1	T2	T3
W(Y := 1)		
Commit		
	Start	
	R(X) → 0	
	R(Y)→ 1	
		W(X:=2)
		W(Z:=3)
		Commit
	R(Z) → 0	
	R(Y) → 1	
	W(X:=3)	
	Commit-Req	
	Abort	

Concurrent updates on Z by T3 not visible
Not first-committer of X
Therefore, T2 is rolled back

각 Transaction은 시작하는 시점에 각자 snapshot을 떠서 processing 을 하고, CC Manager에 Commit-request 하면, (CC Manager가 concurrent transaction들의 update를 비교하여, 허가여부를 결정) 허가를 얻으면 commit (database write)을 진행하다

Read in Snapshot Isolation CC Scheme

- Concurrent updates invisible to snapshot read

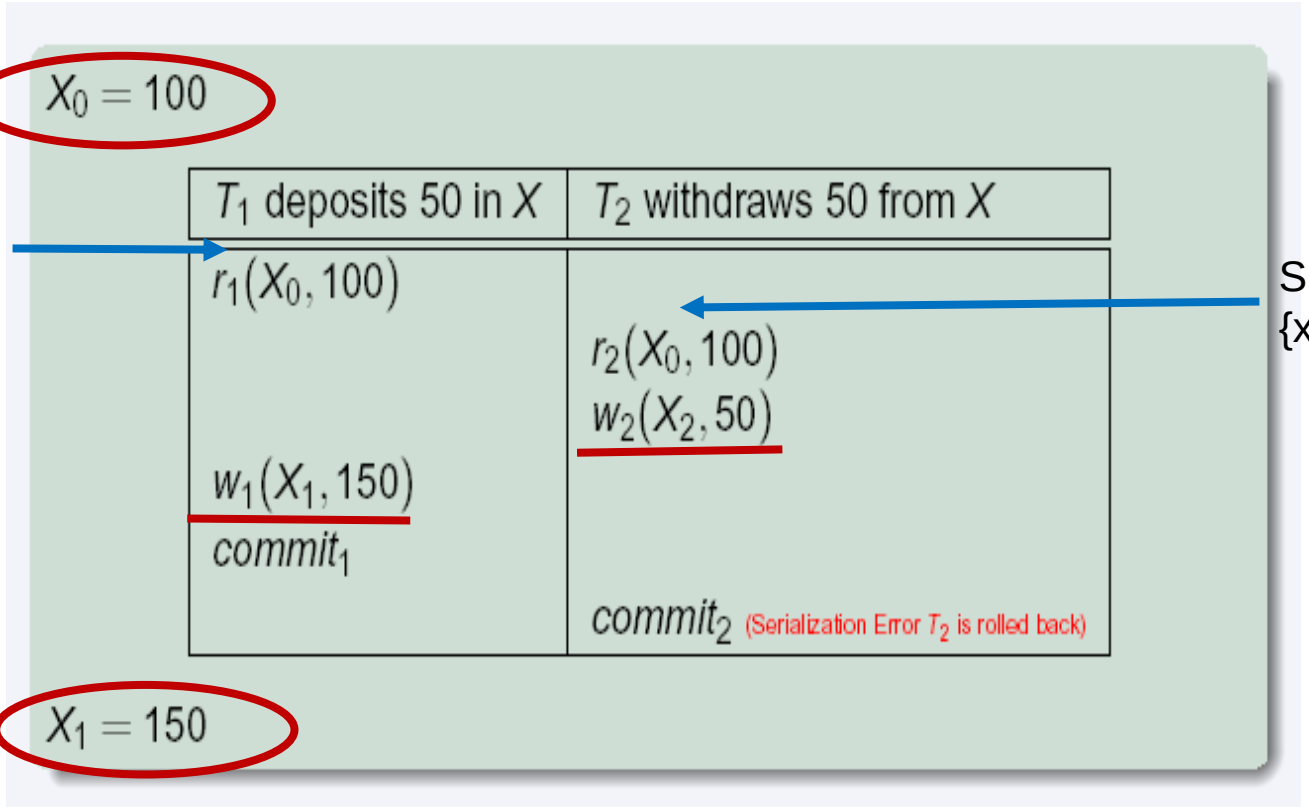


Snapshot isolation 을 사용한 스케줄로, T2가 Write(X)한 것은 T1에 보여지지 않고, T1이 Write(Y)한 것을 T2에 보여지지 않고 있음.
T1, T2의 각각 commit-request에 대해서 cc manager가 허락을 하게 되면 database write이 진행됨. 결과는 T1, T2 순의 serial 스케줄과 동등함

Write in Snapshot Isolation CC Scheme

"First Committer Wins"

참고자료



Snapshot {x0=100}

Snapshot {x0=100}

- If "**First-committer-wins**"가 사용되면 T2의 commit-request가 reject ➔ abort해야 함
- If Variant: "**First-updater-wins**" 가 사용되었다면
  - Check for concurrent updates when write occurs
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

Pros and Cons of Snapshot Isolation CC Scheme

- Pros of Snapshot Isolation
  - Read operations are *never* blocked and also doesn't block other transaction's activities (➔ High Concurrency)
    - Performance of read operations is similar to that of Read Committed
  - Avoids the usual anomalies
    - No dirty read
    - No lost update
    - No non-repeatable read
  - Predicate based selects are repeatable (no phantom anomaly)
- Cons of Snapshot Isolation
  - SI does not always give serializable executions
    - Serializable CC scheme: among two concurrent transactions, one sees the effects of the other
    - Snapshot Isolation: neither sees the effects of the other (commit 하기 전에 확인단계 필요)
  - Result: Integrity constraints can be violated
  - Skew-write anomaly

Snapshot Isolation Anomalies [1/3]

- **Skew-write Anomaly**
  - Initially a snapshot {x = 3 and y = 17}
    - T1: x:= y T2: y:= x
      commit-request commit-request
    - Serial execution: If {T1; T2} ➔ {x = 17, y = 17} or If {T2;T1} ➔ {x =3, y = 3}
    - However if both transactions start at the same time with snapshot {x = 3, y = 17}, T1 and T2 would receive commit permission from CC manager.
      - The result would be x = 17 by T1, y = 3 by T2
      - So, the final database state is { x = 17 , y = 3} // But, this is anomaly
    - **Why this happens?**
    - **X and Y seems "not competing", but are "actually competing"**

Snapshot Isolation Anomalies [2/3]

- Skew-write Anomaly also occurs with inserts
  - Suppose New_Order_Transaction Tno, an order record (order_num, item, client)
    - Step1: Find max_order_no among all orders
    - Step2: insert(max_order_no + 1, item, client) // create an order record
  - SI CC Scheme에서 Tno_1, Tno_2 가 concurrently 수행하고 commit-request
    - Snap-isolation CC manager는 update conflict를 detect 못하므로 전부 허가
    - 모든 concurrent transaction들의 insert의 order_num은 1씩만 증가함
    - But DB에 previous_max라는 variable에서 order_num을 control한다면
      - Step1: previous_max = previous_max + 1
      - Step2: insert(previous_max, item, client) // create an order record
      - 이런식이면 update conflict을 detect하고 Tno_1, Tno_2중에 abort시키면서 previous_max값은 제대로 증가됨
  - 전통적인 serializability을 추구하는 2PL을 썼을때는?

Snapshot Isolation Anomalies [3/3]

- As we saw in the previous two examples
- Serializability is not guaranteed when transactions modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when transactions conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But when it does occur
    - Application developers should be careful about skew-write
    - Update conflict를 제대로 유발하도록 variable을 잘 선정해야 함
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details

- Snapshot isolation은 앞서 언급한 것 같은 anomaly가 드물지만 발생할 수 있기 때문에, 완전한 serializability와 consistency를 요구하는 응용에서는 적합하지 않음

Snapshot Isolation in Oracle and PostgreSQL

- **Warning**: Snapshot Isolation level is used when isolation level is set to serializable, by Oracle and PostgreSQL
  - PostgreSQL's implementation of SI described in Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
  - Neither supports true traditional "(conflict) serializable" execution

- Can sidestep for read-only queries by using **for update** in Oracle and PostgreSQL
  - Locks the data which is read, preventing concurrent updates
  - This can evade the phantom phenomenon
  - **select max**(orderno) **from** orders **for update**
    1. read value into a local variable maxorder
    2. insert into orders (maxorder+1, …)

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

Insert a Tuple and Delete a Tuple while Read/Write Transactions [1/2]

- If 2PL is used in the situation of title :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple ➔ (DB에 존재하지 않는 new tuple에 lock을 거는 방법은?)
- Insertions and deletions can lead to the **phantom phenomenon**
  - A transaction T1 that scans a relation
    - Step1: Sum = sum of balances of all accounts in Perryridge
    - Step2: Num = number of accounts in Perryridge
    - Step3: Avg = Sum / Num
  - And a transaction T2 that inserts a tuple in the relation Perryridge
    - insert a new account at Perryridge
  - T1 and T2 (conceptually) conflict in spite of not accessing any tuple in common
  - If only tuple locks are used, non-serializable schedules can result
    - T1의 Step1과 Step2 사이에 T2가 수행되었다면 Step3의 Avg값은 serial schedule에서는 나올수 없는 이상한 값

- The phantom issue was identified in Eswaran, Gray, Lorie, and Traiger, "The Notion of Consistency and Predicate Locks in a Database System" CACM 1976

Insert a Tuple and Delete a Tuple while Read/Write Transactions [2/2]

- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information
  - Somehow, some information should be locked!
- One naïve solution:
  - (A) Transactions scanning the relation acquire a S-lock on the entire relation
  - (B) Transactions inserting or deleting a tuple acquire a X-lock on the data item
- Above protocol provides very low concurrency for insertions/ deletions

- Solutions (Phantom Phenomenon을 비껴가는)
  - Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets
    - 아직 insert가 안된 tuple T에 lock을 걸수는 없지만, insert에 관여하는 B+ index에 X-lock을 걸고, read-only transaction들도 꼭 B+ index를 navigat하면서 S-lock을 걸면 update conflict을 유발기킬수 있다
  - Predicate locking protocol: (A)의 query에 있는 predicate과 (B)의 내용이 conflict이 있으면 (A)와 (B)를 interleaving시키지 않는다

Phantom Read in Weak Consistency Example

참고자료

- Phantom현상은 serializibility consistency level보다 약한 consistency level에서도 발생가능하다.
- Repeatable Read Consistency Level을 구현한 Snapshot Isolation을 쓰는 상황이라면 아래와 같은 T1, T2의 interleaving의 결과는 잘못된 결과이다.
  - Snapshot Isolation에서는 한 transaction내에서 같은 query를 꼭 같은값을 return해야 한다

- Repeatable Read, Read Committed, Read uncommitted 모두에서 팬텀 read 발생 가능함
  - Serializable level에서는 아래상황은 그냥 상식적으로 처리된다

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```
/* Query 2 */
INSERT INTO users VALUES ( 3, 'Bob', 27 );
COMMIT;
```

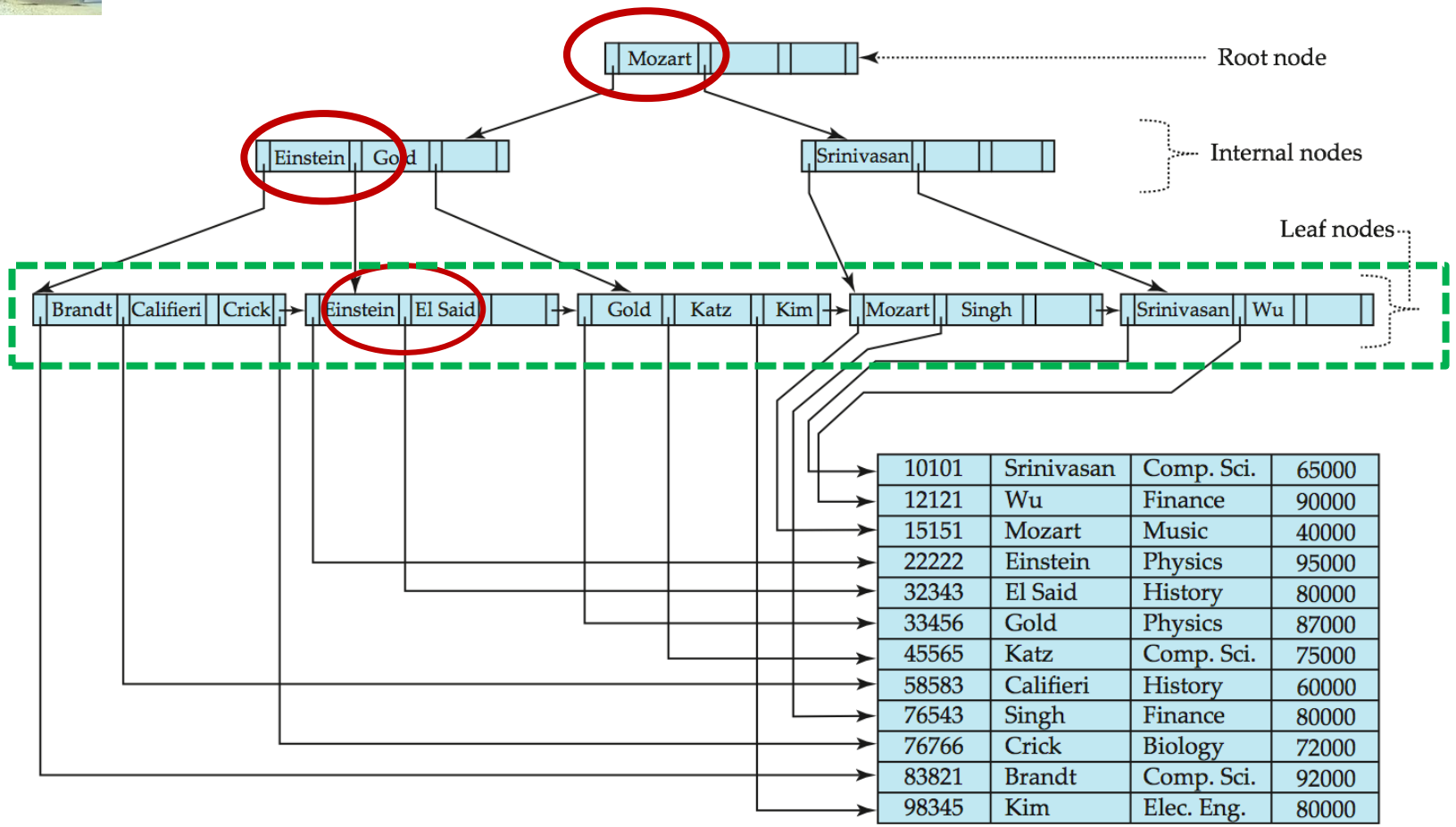
```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
COMMIT;
```

Index Locking Protocol

- Bayer and Schkolnick, “Concurrency of Operating on B-trees”, Acta Infomatica, 1977
- Index locking protocol:
  - Every relation must have at least one index
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode (read-only transaction도 꼭 B+ tree를 navigate)
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g., for a range query, no tuple in a leaf is in the range)
  - A transaction T_i that inserts, updates or deletes a tuple t_i in a relation r
    - must update all indices to r
    - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the 2PL protocol must be observed
- Guarantees that phantom phenomenon won’t occur in the index locking protocol

- Index Locking Protocol은 B tree Index nodes에서의 CC를 의미하는것이 아님
- B tree Index nodes에서 lock conflict를 minimize하는 것은 Chap 15.10에서 cover

Index Locking Protocol Example



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Suppose
T1: transaction computing average salary
T2: transaction inserting a new tuple (44427, Ebra, Com. Sci., 80000)

Lock을 red circle에 걸면 Locking area도 최소화하고, Phantom phenomenon도 피함

Predicate Locking

참고자료

- Eswaran, Gray, Lorie, and Traiger, “The Notion of Consistency and Predicate Locks in a Database System” CACM 1976

- Locking Predicates
  - S-Lock the predicate in a where-clause of a SELECT ex) “salary > 90000”
  - X-lock the predicate in a where clause of an UPDATE, INSERT or DELETE

- Conflict decision
  - A lock can’t be granted if a conflicting lock is held already
  - For predicates, a Lock on P1 by T1 conflicts with Lock on P2 by T2 if
    - Locks are not both S-mode
    - T1 different from T2
    - Some record r could exist in the schema such that P1(r) and P2(r)

- Never implemented in any system

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

Weak Levels of Consistency [1/2]

- **Weaker consistency** 통해서 **More concurrency**를 추구하는 시도
- **Degree-two consistency :** Gray, Lorie, Putzolu, "Granularity of Locks and Degree of Consistency in a Shared Data Base", VLDB 1975
  - 2PL의 변형
    - locks may be acquired at any time
    - S-locks may be released at any time, X-locks must be held till end of transaction (Thus, avoids cascading aborts)
  - Serializability is not guaranteed in DB transaction level
    - The application level must ensure that no erroneous database state will occur

Serializable schedule은 아니지만 큰문제는 없는 schedule

Figure 15.20 Nonserializable schedule under degree-2 consistency

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)	

2PL이 사용됐다면 아래와 같은 serializable schedule이 생성

T32	T33
Lock-s (Q)	
Read(Q)	
…	
Read(Q)	
Unlock(Q)	
	Lock-x(Q)
	Read(Q)
	Write(Q)
	Unlock(Q)

Weak Levels of Consistency [2/2]

- **Cursor stability** (IBM DB2)
  - A form of degree-two consistency for programs written in host language
  - DBMS가 아니라 Application program에서 relation에 cursor를 두고 tuple들에 반복적인 작업을 하는 경우에, the entire relation을 lock하는것이 아니라
    - For reads, each tuple is S-locked, read, and the S-lock is immediately released
    - X-locks are held until the transaction commits
  - Not 2PL!, so serializability is not guaranteed, but the performance improves

- 나중에 SQL의 weak levels of consistency로 정리가 됨

Weak Levels of Consistency in SQL

- SQL allows several levels of consistency (ANSI SQL Isolation levels)
  - **Serializable**: used to be the default before 1990
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read
- Some modern database systems do not ensure serializable schedules by default
  - Oracle and PostgreSQL support serializability (but it means snapshot isolation)
  - MS SQL-server support read-committed by default
  - Consistency level can be reset by the command (**set isolation level serializable)**

** Commercial DBMS에서 user에게 consistency level을 선택하게 하고 concurrency와 trading off

상용DB의 Consistency level [1/2]

참고자료

Oracle

Isolation Level	Description
Read committed (default)	This is the default transaction isolation level. Each query executed by a transaction sees only data that was committed before the query (not the transaction) began. An Oracle Database query never reads dirty (uncommitted) data. Because Oracle Database does not prevent other transactions from modifying the data read by a query, that data can be changed by other transactions between two executions of the query. Thus, a transaction that runs a given query twice can experience both nonrepeatable read and phantoms.
Serializable	Serializable transactions see only those changes that were committed at the time the transaction began, plus those changes made by the transaction itself through `INSERT`, `UPDATE`, and `DELETE` statements. Serializable transactions do not experience nonrepeatable reads or phantoms.
Read-only	Read-only transactions see only those changes that were committed at the time the transaction began and do not allow `INSERT`, `UPDATE`, and `DELETE` statements.

In fact, this is snapshot isolation

상용DB의 Consistency level [2/2]

참고자료

Microsoft SQL Server (ver. 2005, 2008, 2008R2, 2012, 2014)

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Read committed using row versioning	No	Yes	Yes
Repeatable read	No	No	Yes
Snapshot	No	No	No
Serializable	No	No	No

default

PostgreSQL

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

default

In fact, this is snapshot isolation

Concurrency Control Across User Interactions [1/2]

- Long duration transactions with user interactions
  - Airline seat selection transaction
  - VLSI design transaction

- If 2PL is used, the entire seats or the large area of VLSI circuit would be S-locked
- If Time-stamp CC or Validation CC are used, transaction abort 가 발생하면 damage가 크다
- If snapshot isolation was used, transaction이 commit한 이후에도 다른 concurrent transactions들이 commit을 안한 상태이면 계속 update정보를 계속 기억하고 있어야 한다
- What if split the long transaction with 2 or more transactions?
  - The 1st transaction A would read the seat availability
  - The 2nd transaction B would complete the allocation of the selected seat
  - ➔ A 와 B사이에 다른 transaction이 B가 선택한 seat을 가져가 버렸다면 lost-update problem이 생김

- In summary, existing CC는 long duration transaction에 fit하지 않는다!

Concurrency Control Across User Interactions [2/2]

- Many applications need transaction support across user interactions
  - Can't use the regular locking CC schemes
  - Don't want to reserve database connection per user
- Application level concurrency control
  - Each tuple has a version number
  - Transaction notes version number when reading tuple
    - **Select** r.balance, r.version **into** :A, :version
      from r **where** acctId =23
  - When writing tuple, check that current version number is same as the version when tuple was read
    - **Update** r **set** r.balance = r.balance + :deposit
      where acctId = 23 **and** r.version = :version
- Equivalent to **optimistic concurrency control without validating read set**
- Used internally in Hibernate ORM system, and manually in many applications
- Version numbering can also be used to support first committer wins check of snapshot isolation
  - Unlike SI, reads are not guaranteed to be from a single snapshot

Chapter 15: Concurrency Control

- Lock-Based Protocols
- Deadlock Handling
- Multiple Granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Snapshot Isolation
- Insert and Delete Operations
- Weak Levels of Consistency
- Concurrency in Index Structures

Concurrency in Index Structures [1/2]

- Indices are unlike other DB items in that their only job is to help in accessing data
- Index-structures are typically accessed very often, much more than other DB items
  - Treating index-structures like other DB items, e.g., by 2PL of index nodes can lead to low concurrency
- There are several index concurrency protocols where locks on internal nodes are released early, and not in 2PL fashion
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained
    - In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node

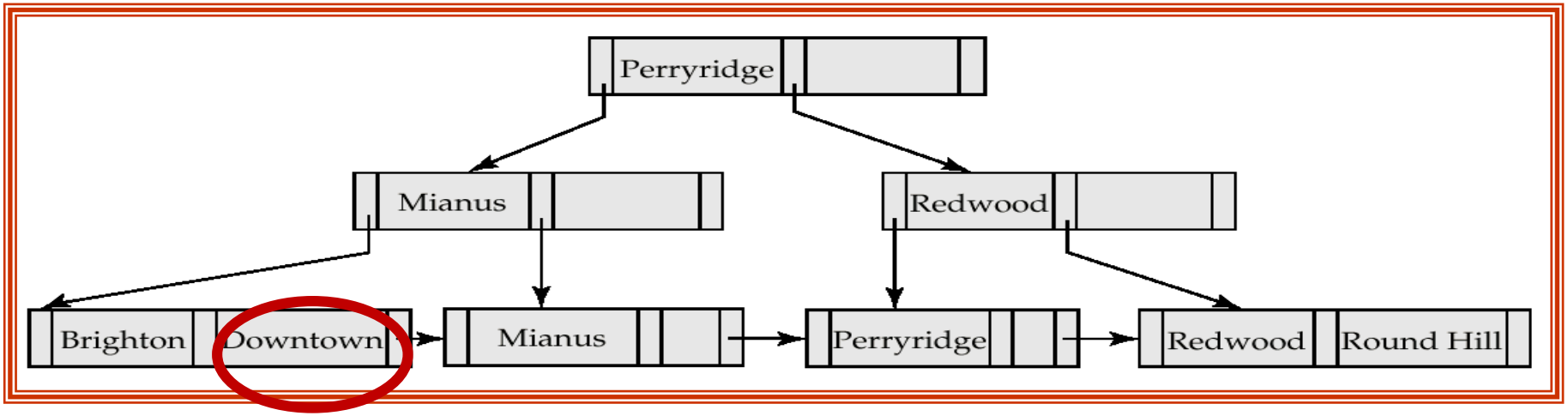
앞에서 Phantom phenomenon을 이야기할때는 read transaction과 insert/delete transaction간의 문제해결에 locking index protocol을 사용.
여기서는 여러 개의 insert/delete transaction들이 B tree index를 함께 update하는 상황에서 conflict을 줄이려고 노력하는 Index cc protocol를 다룸.

Concurrency in Index Structures [2/2]

- Bayer and Schkolnick, “Concurrency of Operating on B-trees”, Acta Infomatica, 1977
- **Crabbing protocol (**instead of 2PL on the nodes of the B$^+$-tree)
  - During search/insertion/deletion :
    - First lock the root node in S mode
    - After locking all required children of a node in shared mode, release the lock on the root node
  - During insertion/deletion
    - upgrade leaf node s-locks to X mode
    - Perform inserts or deletes the key value
    - When splitting or coalescing requires changes to a parent, lock the parent in X mode
      - Recursively keep going this step
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Solution: Abort and restart the search, without affecting transaction
- Better protocols such as the B-link tree protocol are available in Section 15.10
  - Intuition: release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

Crabbing Protocol for Index Update [1/2]

참고자료



[Search] Downtown을 search 한다면

① 루트 노드에 S-lock
② 탐색하는 자식 노드들에도 S-lock
③ 자식 노드에 락을 걸면 부모 노드의 락은 해제
④ 단말 노드에 도달할 때까지 위 작업을 반복함

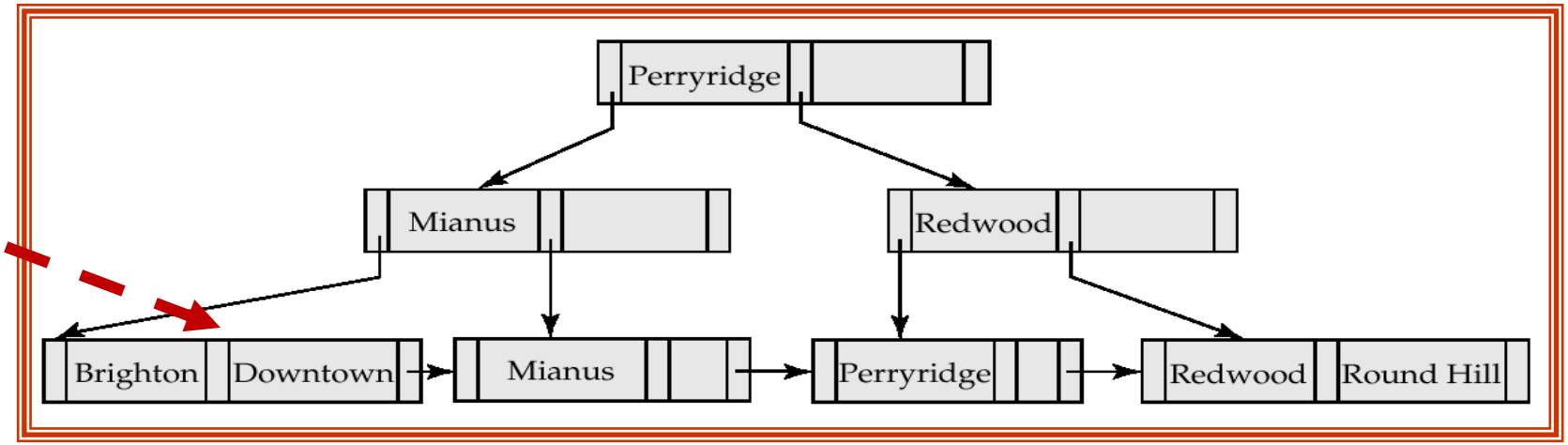
[Insert/Delete] clearview를 삽입하려고 한다면

① 단말노드에 도착할 때까지는 탐색방법과 동일
② 단말 노드에 도착하면 X-lock을 걸고 키 값을 삽입/삭제
if) 노드를 분할/재분배 해야 될 경우
(1) 해당 노드의 부모 노드에 X-lock
(2) 분할/재분배가 위로 전파될 경우, 같은 방법으로 부모 노드에 X-lock
(3) 분할/재분배 작업이 끝나면 락 해제

Crabbing Protocol for Index Update [2/2]

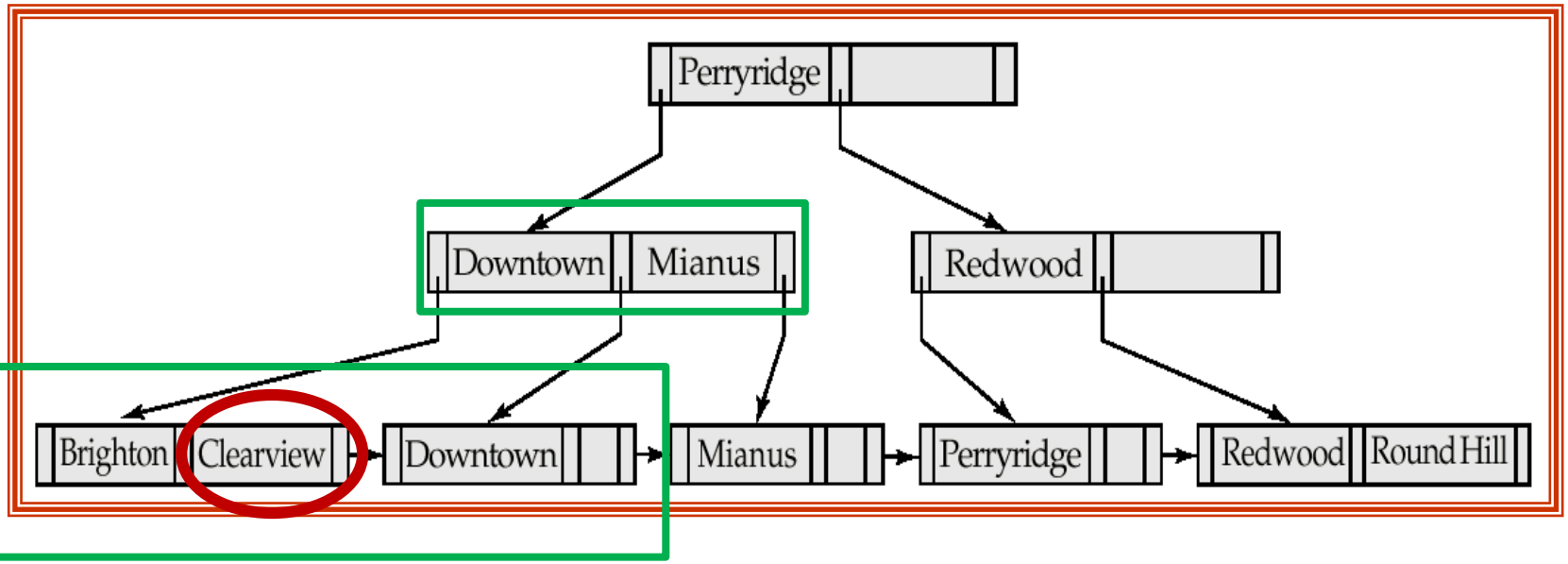
참고자료

Clearview
삽입시도



Another
search

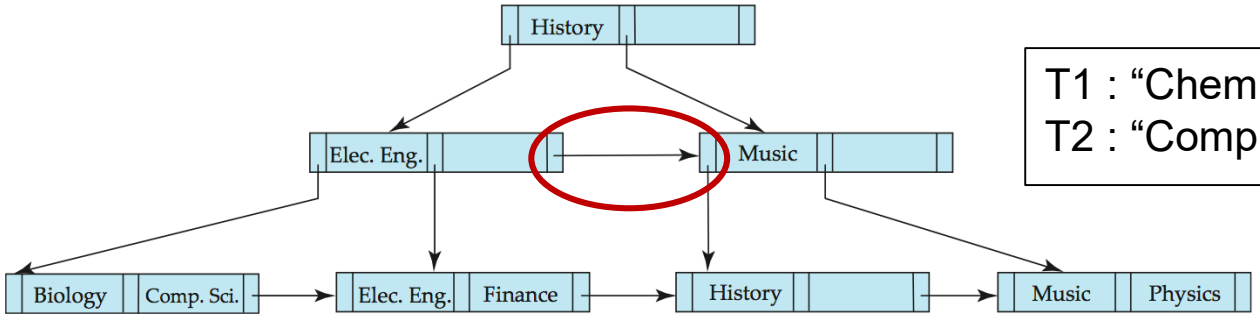
Update
작업



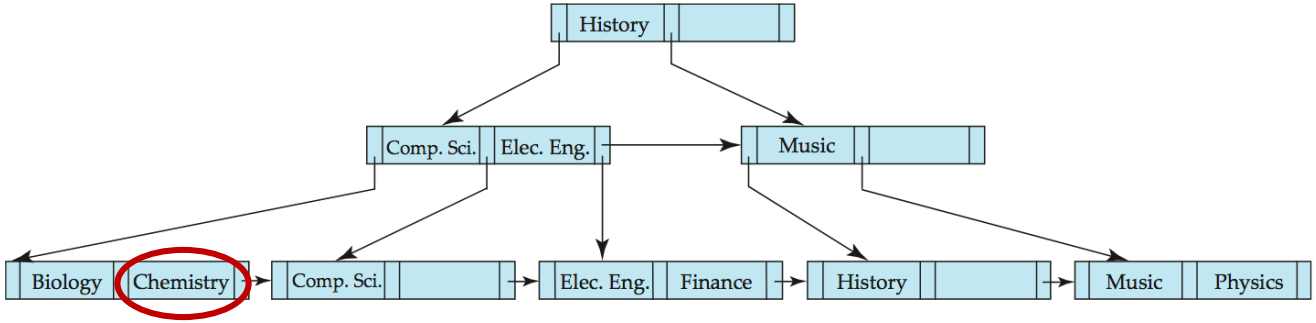
B-link Tree Locking Protocol Example

참고자료

- Lehman and Yao [1981]
- B-link tree is similar to B+ tree except
  - Every node has a right sibling pointer
  - Update transactions release lock on parent before acquiring lock on child
- B+ tree의 update상황보다 deadlock이 많이 줄어들고 Concurrency가 높아진다
  - Details in Page 706-707



T1 : “Chemistry” 삽입
T2 : “Comp.Sci” 탐색



Next-Key Locking

- Index-locking protocol to prevent phantoms required locking entire index leaf node
  - Can result in poor concurrency if there are many inserts through many leaf nodes
    - Lead node가 cover하는 data record들 전부를 lock하게 되므로
- Alternative: Next-Key Locking (Index leaf node가 아니라, 실제 data record에 locking)
  - Mohan [1990a] “ARIES/KVL” (Key-Value Locking)
  - For an index lookup (using shared lock S):
    - Lock (shared lock S) all values that satisfy index lookup
      - Single key value for a point lookup, or key values that fall in lookup range
    - Also lock next key value in index
      - The key value just greater than the last key value that was within the range
  - For insert/delete/update (using exclusive lock X):
    - Lock (exclusive lock X) the value V that is inserted
    - Also lock the next key which is greater than V
- Ensures that range queries will conflict with inserts/deletes/updates
  - Regardless of which happens first, as long as both are concurrent
  - next-key는 이미 존재하므로 충돌이 발생하면 감지가능함

Next-Key Locking Example

참고자료

Q1: Key값이 30보다 작은 data를 읽는 query
Q2: Insert a record having a key value 35

Index-locking protocol을 썼다면 Q1을 처리하면서 분홍색지역을 cover하는 index leaf node (10-40)에 lock을 걸어서 Q2와 충돌 유발

Next-Key locking을 썼다면
Q1이 30을 읽을때에 next-key 40에도 S-lock걸고
Q2도 35를 insert를 하면서 next-key 40에 X-lock 걸었다면 Q1과 Q2의 충돌이 유발