Advanced DB

CHAPTER 16
CONCURRENCY CONTROL

Chapter 16: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Deadlock Handling
- Insert and Delete Operations
- Concurrency in Index Structures

Implementation of Isolation

- Schedules must be conflict (or view serializable), and recoverable (for database consistency)
  - and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Two modes :
  1. *exclusive (X) mode*: both read and write
     (lock-X instruction)
  2. *shared (S) mode*: only read
     (lock-S instruction)
- Lock requests are made to concurrency-control manager
- Transaction can proceed only after request is granted.

Granting of Locks

- 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 lock(s) already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- If any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Example

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

- 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.

Two-Phase Locking Protocol

- Locking Protocol
  - A set of rules followed by all transactions while requesting and releasing locks
  - Locking protocols restrict the set of possible schedules.
- 2PL
  - Phase 1: Growing Phase
    - transaction may obtain locks
      - can acquire a **lock-S** or **lock-X** on item
      - can convert a **lock-S** to a **lock-X** (**upgrade**)
    - transaction may not release locks
  - Phase 2: Shrinking Phase
    - transaction may release locks
      - can release a **lock-S** or **lock-X**
      - can convert a **lock-X** to a **lock-S** **(downgrade**)
    - transaction may not obtain locks

Example

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)

Features of 2PL

- Serializability: the protocol assures conflict serializability
  - It can be shown that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).
  - There can be conflict serializable schedules that cannot be obtained if two-phase locking is used
- Deadlocks: Two-phase locking *does not* ensure freedom from deadlocks
  - starvation also possible
- Cascading rollback: is possible under two-phase locking

Strict / Rigorous 2PL

- Strict 2PL
  - To avoid cascading roll-back
  - A transaction must hold all its exclusive locks until it commits/aborts

- Rigorous 2PL
  - *all* locks are held until commit/abort
  - transactions can be serialized in the order in which they commit

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system.
  - Older transaction T_i has smaller time-stamp than newer transaction T_j

$$\mathrm{TS}(T_i) < \mathrm{TS}(T_j).$$

- 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.

Timestamp-Ordering Protocol

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

1. Transaction T_i issues read(Q)
   - If TS(T_i) < **W**-timestamp(Q)
     - reject **read** operation, and T_i is rolled back.
     - Since this means T_i needs to read a value of Q that was already overwritten.
   - If TS(T_i)≥ **W**-timestamp(Q)
     - execute **read** operation
     - set R-timestamp(Q) = max(R-timestamp(Q), TS(T_i))

Timestamp-Ordering Protocol (Cont.)

2. Transaction T_i issues write(Q).
   - If TS(T_i) < R-timestamp(Q)
     - reject **write** operation, and T_i is rolled back.
     - Since the value of Q that T_i is producing was needed previously, and the system assumed it would never be produced.
   - If TS(T_i) < W-timestamp(Q)
     - reject **write** operation, and T_i is rolled back.
     - Since T_i is attempting to write an obsolete value of Q.
   - Otherwise
     - execute **write** operation
     - set W-timestamp(Q) = TS(T_i)

Example - Timestamp-Ordering 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($A + B$)

$TS(T_{14}) < TS(T_{15})$

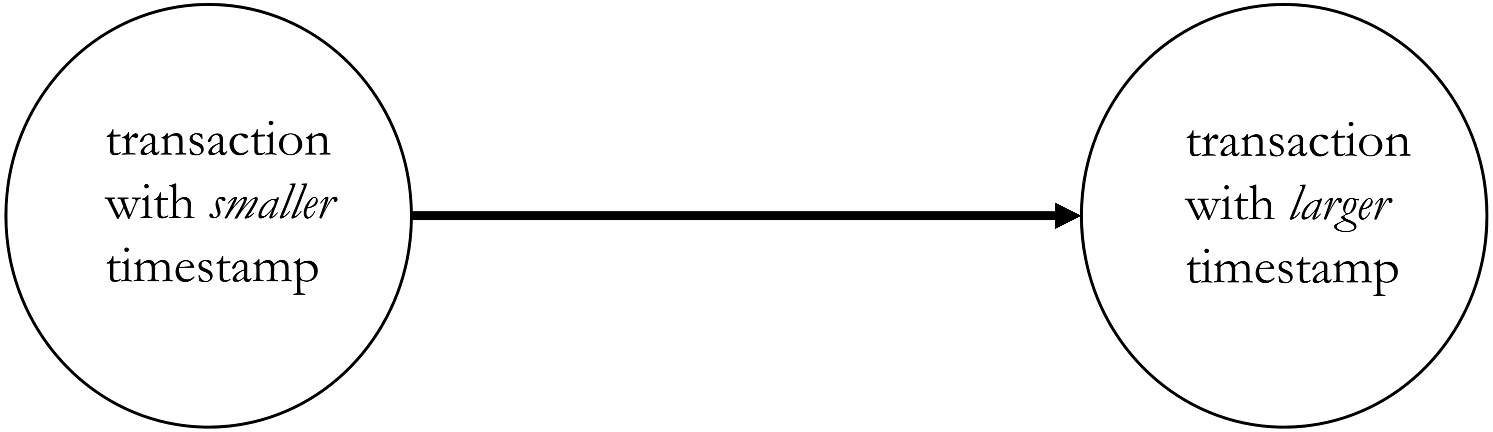
Example - Timestamp-Ordering Protocol

Timestamp: 1 2 3 4 5

T_1	T_2	T_3	T_4	T_5
read(*Y*)				
	read(*Z*)			
			read(*X*)	
		write(*Y*)		
				read(*Y*)
				read(*Z*)
			write(*X*)	
	read(*X*)			
read(*Z*)				
		write(*Z*)		
				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:



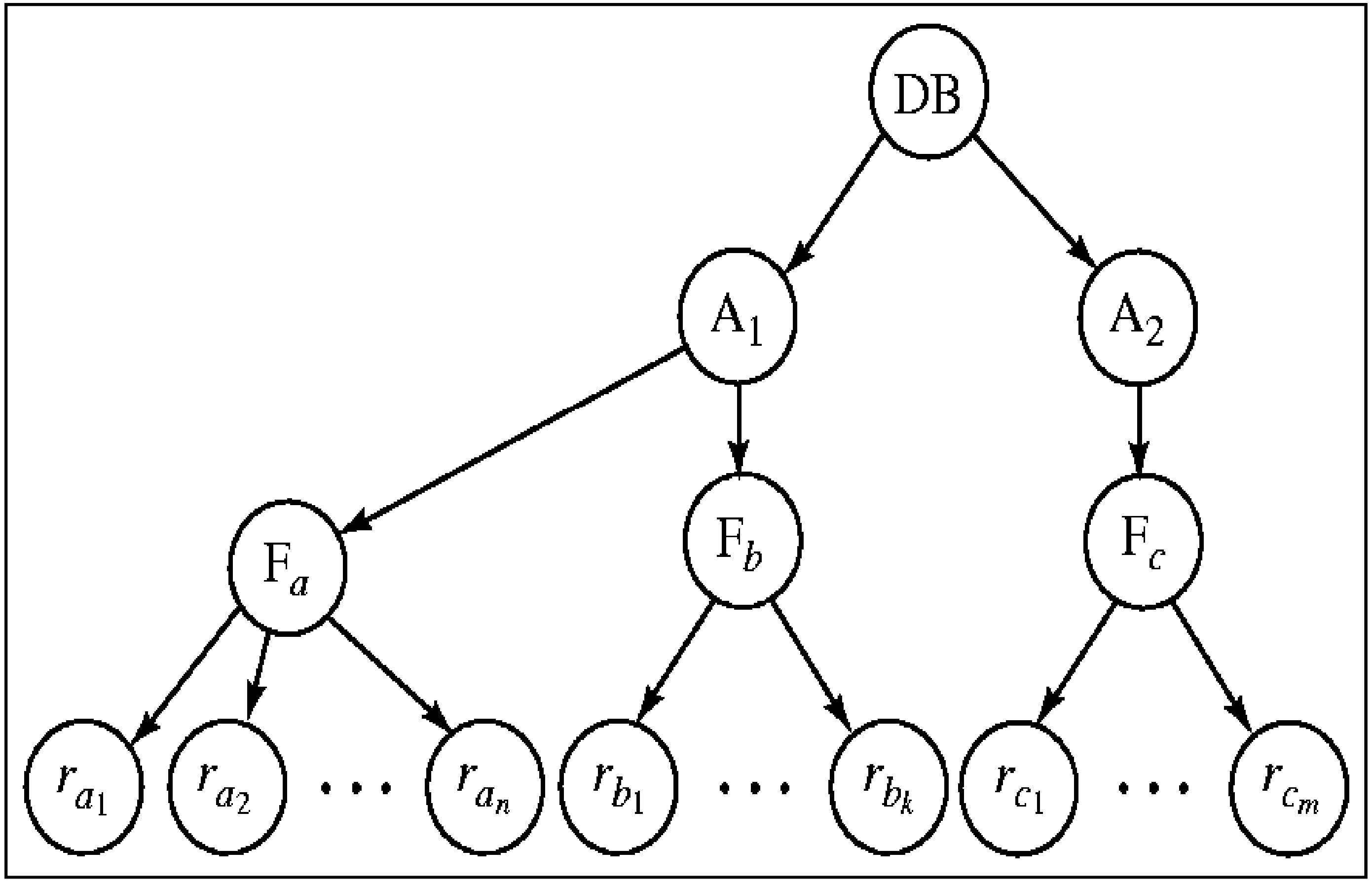
Thus, there will be no cycles in the precedence graph

- 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.

Multiple Granularity

- 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
- An *explicitly* lock on a node implies *implicit* locks on 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

Example - Granularity Hierarchy



Sample hierarchy: database => *area* => *file* => *record*

Intention Lock Modes

- Three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): explicit shared locking at a lower level
  - ***intention-exclusive*** (IX): 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 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

- The compatibility matrix for all lock modes including intention locks

	IS	IX	S	S IX	X
IS	O	O	O	O	×
IX	O	O	×	×	×
S	O	×	O	×	×
S IX	O	×	×	×	×
X	×	×	×	×	×

Multiple Granularity Locking Scheme

- T_i can lock node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. Root of the tree must be locked first
  3. Q can be locked by T_i in S or IS mode only if T_i currently holds IX or IS mode lock on the parent of Q
  4. Q can be locked by T_i in X, SIX, or IX mode only if T_i currently holds IX or SIX mode lock on the parent of Q
  5. T_i can lock a node only if it has not previously unlocked any node (i.e., observe is 2PL).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Multiple Granularity Locking Scheme

- Enhances concurrency and reduces lock overhead
  - Mix of short transactions that access few data items and long transactions that access entire tables.
- Ensures serializability
- Is not deadlock free
- Example
  - T18: read(r_{a2})
  - T19: write(r_{a9})
  - T20: read(F_a)
  - T21: read(DB)

Insert and Delete Operations

- If two-phase locking is used :
  - 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
- Insertions and deletions can lead to the **phantom phenomenon**.
  - T_{29}: **select sum**(*balance*) **from** *account* **where** *branch-name*=‘Perryridge’
  - T_{30}: **insert into** *account* **values** (‘A201’, ‘Perryridge’, 1000)
  - may conflict *in spite of not accessing any tuple in common.*
- If only tuple locks are used, non-serializable schedules can result:

  T_{29} may not see the new account, yet may be serialized to come after the T_{30}

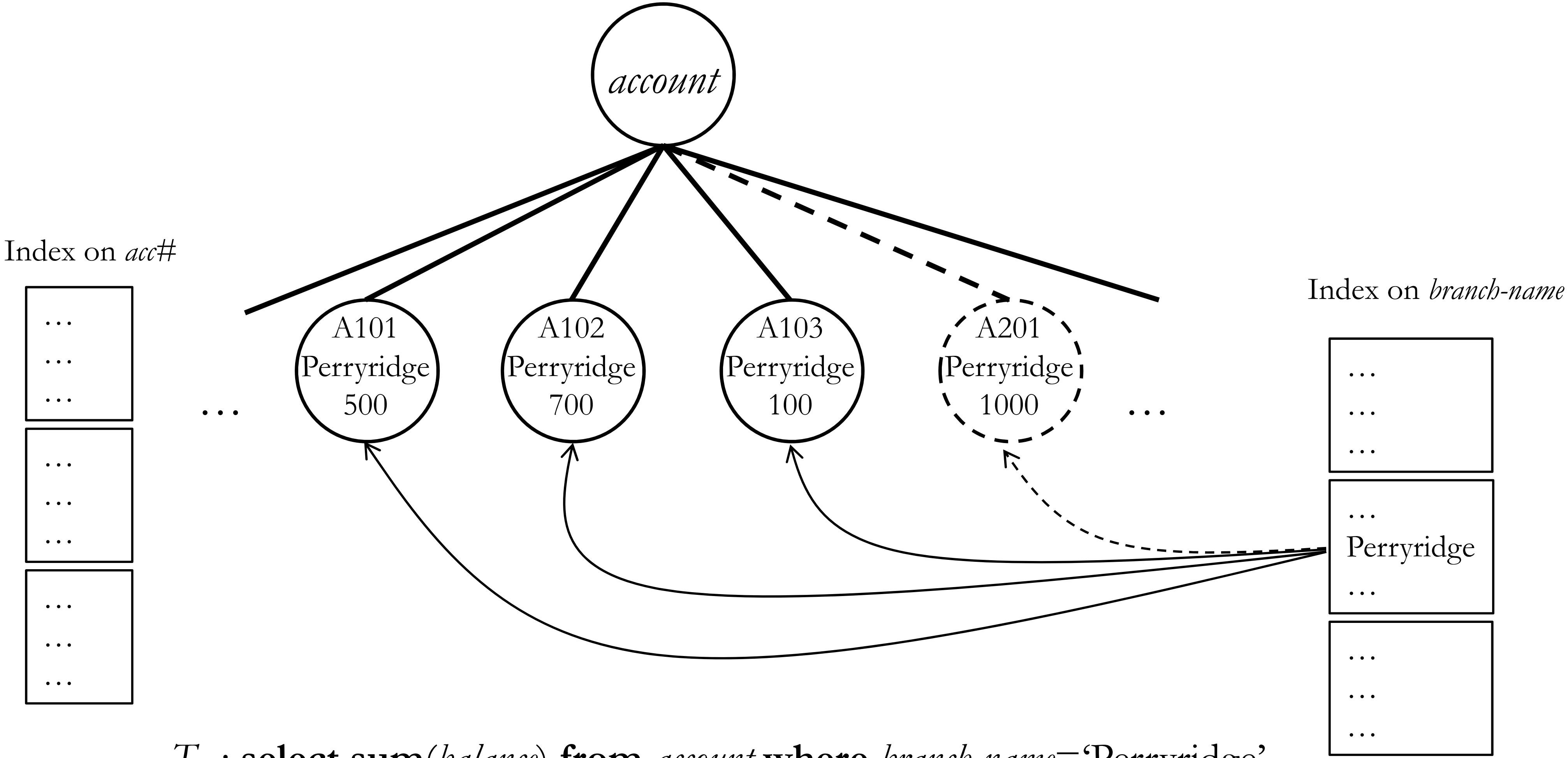
Insert and Delete Operations (Cont.)

- Can multiple granularity locking protocol be a solution?
  - How? Or why not?
- Observation
  - The scan transaction must use (read) information that indicates what tuples the relation contains,
  - while the insert transaction updates the same information.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item.
  - (Note: locks on the data item do not conflict with locks on individual tuples)
- Above protocol provides very low concurrency for insertions/deletions.

Index Locking Protocol

- Every relation must have at least one index.
- A transaction T_i can access tuples of a relation only after first finding them through one or more of the indices.
- A transaction T_i that performs a lookup must lock all the index buckets that it accesses, in S-mode.
- A transaction T_i may not insert a tuple t_i into a relation r without updating all indices to r.
- T_i must perform a lookup on every index to find all index buckets that could have possibly contained a pointer to tuple t_i, had it existed already, and obtain locks in X-mode on all these index buckets. T_i must also obtain locks in X-mode on all index buckets that it modifies.
- The rules of the two-phase locking protocol must be observed

Index Locking Protocol (cont.)



T_{29}: **select sum**(*balance*) **from** *account* **where** *branch-name*='Perryridge'

T_{30}: **insert into** *account* **values** ('A201', 'Perryridge', 1000)

Weak Levels of Consistency

- **Degree-two consistency:** S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

T_3	T_4
lock-S(Q)	
read(Q)	
unlock(Q)	
	lock-X(Q)
	read(Q)
	write(Q)
	unlock(Q)
lock-S(Q)	
read(Q)	
unlock(Q)	

Concurrency in Index Structures

- Indices are unlike other database items in that
  - their only job is to help in accessing data.
  - they are typically accessed very often, much more than other database items
- Treating index-structures like other database items leads to low concurrency
  - Two-phase locking on an index may result in transactions executing practically one-at-a-time
- It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
  - 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

Concurrency in Index Structures (Cont.)

- There are index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion
- **Crabbing Protocol** (for nodes of the B^+-tree index)

  During search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node.
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- can cause excessive deadlocks
  - Better protocols are available

END OF CHAPTER 16