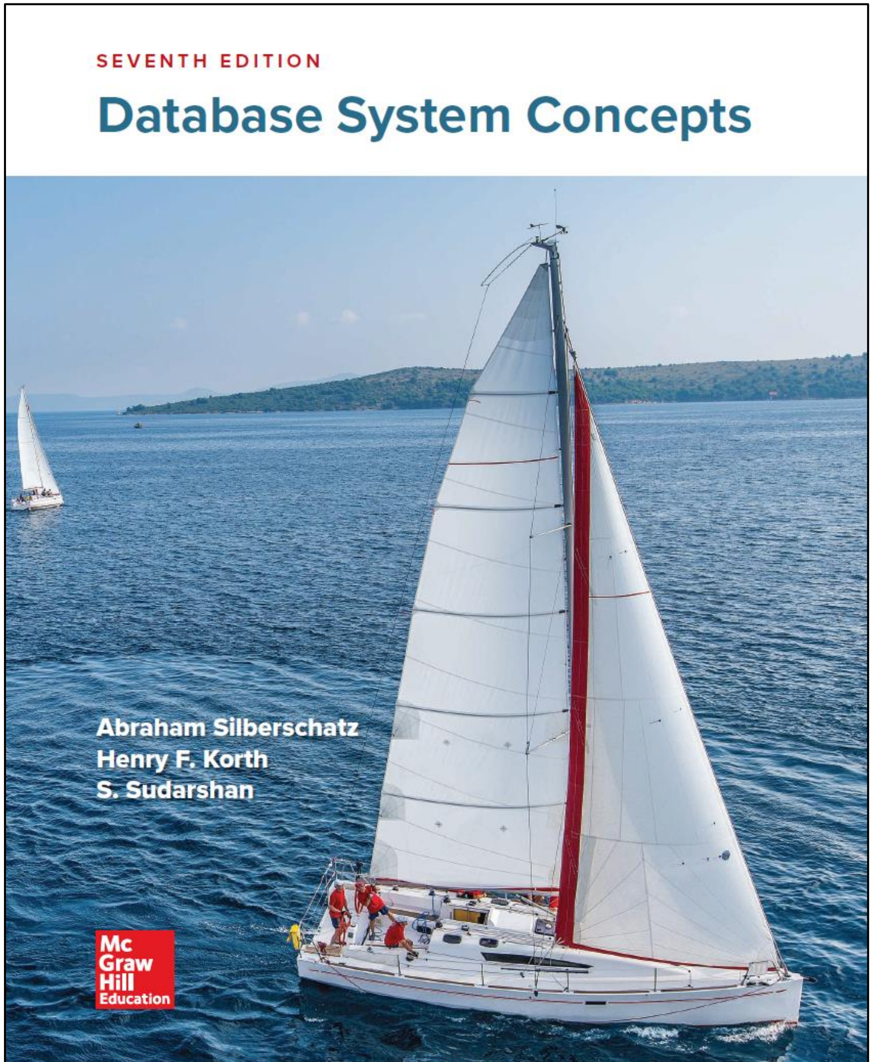
Data Administration in Information Systems

Transactions and concurrency

Transactions and concurrency



Chapter 16 Query Optimization

16.1 Overview 743
16.2 Transformation of Relational Expressions 747
16.3 Estimating Statistics of Expression Results 757
16.4 Choice of Evaluation Plans 766
16.5 Materialized Views 778
16.6 Advanced Topics in Query Optimization 783
16.7 Summary 787
Exercises 789
Further Reading 794

PART SEVEN ■ TRANSACTION MANAGEMENT

Chapter 17 Transactions

17.1 Transaction Concept 799
17.2 A Simple Transaction Model 801
17.3 Storage Structure 804
17.4 Transaction Atomicity and Durability 805
17.5 Transaction Isolation 807
17.6 Serializability 812
17.7 Transaction Isolation and Atomicity 819
17.8 Transaction Isolation Levels 821
17.9 Implementation of Isolation Levels 823
17.10 Transactions as SQL Statements 826
17.11 Summary 828
Exercises 831
Further Reading 834

Chapter 18 Concurrency Control

18.1 Lock-Based Protocols 835
18.2 Deadlock Handling 849
18.3 Multiple Granularity 853
18.4 Insert Operations, Delete Operations, and Predicate Reads 857
18.5 Timestamp-Based Protocols 861
18.6 Validation-Based Protocols 866
18.7 Multiversion Schemes 869
18.8 Snapshot Isolation 872
18.9 Weak Levels of Consistency in Practice 880
18.10 Advanced Topics in Concurrency Control 883
18.11 Summary 894
Exercises 899
Further Reading 904

Chapter 19 Recovery System

19.1 Failure Classification 907
19.2 Storage 908
19.3 Recovery and Atomicity 912
19.4 Recovery Algorithm 922
19.5 Buffer Management 926
19.6 Failure with Loss of Non-Volatile Storage 930
19.7 High Availability Using Remote Backup Systems 931
19.8 Early Lock Release and Logical Undo Operations 935
19.9 ARIES 941
19.10 Recovery in Main-Memory Databases 947
19.11 Summary 948
Exercises 952
Further Reading 956

Implementation of Isolation Levels

- Locking
  - Lock on entire database vs. lock on items
  - How long to hold lock?
  - Shared vs. exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a "snapshot" of the database

Lock-Based Protocols

- 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 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.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

T_2: **lock-S**(A)
read(A)
unlock(A)

lock-S(B)
read(B)
unlock(B)

display($A+B$)

- Locking as above is not sufficient to guarantee serializability

Schedule With Lock Grants

- This schedule is not serializable
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

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_1)
read(A) $A := A + 50$ write(A) unlock(A)		

Schedule With Lock Grants (Cont.)

- Grants will be omitted in the next slides
  - Assume grant happens just before the next instruction in the transaction

T_1	T_2
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
unlock(B)	
	lock-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display($A + B$)
lock-X(A)	
read(A)	
$A := A + 50$	
write(A)	
unlock(A)	

Deadlock

- 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)	

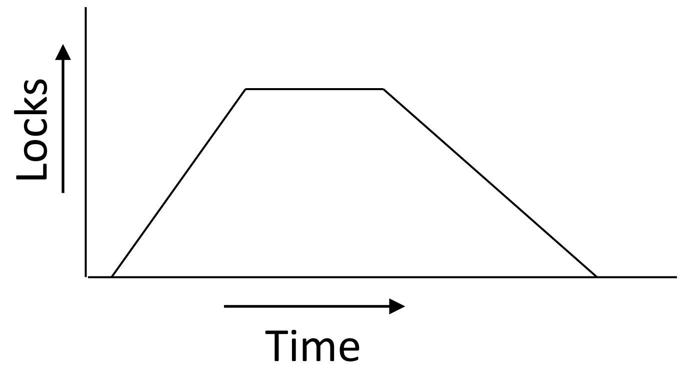
- Neither T_3 nor T_4 can make progress
  - executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To break the deadlock, one of T_3 or T_4 must be rolled back and its locks released.

Deadlock (Cont.)

- The potential for deadlock exists in most locking protocols.
- **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.

The Two-Phase Locking Protocol

- 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)

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking does not prevent **deadlocks**
- Extensions to basic two-phase locking needed to ensure **recoverability** and avoid **cascading rollbacks**
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures **recoverability** and avoids **cascading rollbacks**
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement **rigorous two-phase locking** but refer to it simply as **two-phase locking**

Locking Protocols

- Given a locking protocol (such as two-phase locking)
  - A schedule *S* is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable

Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing 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**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

Automatic Acquisition of Locks

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

if T_i has a lock on D
then
    read(D)
else begin
    if needed, wait until no other transaction has a **lock-X** on D
    grant T_i a **lock-S** on D
    read(D)
  end

Automatic Acquisition of Locks (Cont.)

- The operation **write**_(D)_ is processed as:

  if T_i has a **lock-X** on *D*
  then
  write(*D*)
  else begin
  if needed, wait until no other transaction has any lock on *D*
  if T_i has a **lock-S** on *D*
  then
  upgrade lock on *D* to **lock-X**
  else
  grant T_i a **lock-X** on *D*
  write(*D*)
  end;

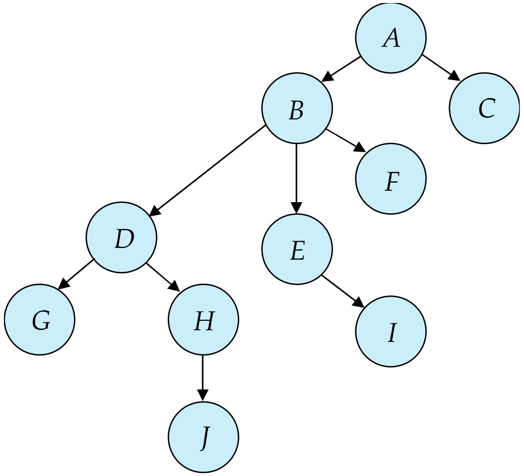
- All locks are released after commit or abort

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ 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*.
- The **tree-protocol** is a simple kind of graph protocol

Tree Protocol

- Only exclusive locks are considered
- The first lock may be on any data item
- Subsequently, a data item can be locked only if its parent is currently locked by the same transaction
- Data items may be unlocked at any time
- A data item that has been locked and unlocked cannot be subsequently re-locked by the same transaction



Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict **serializability** as well as freedom from **deadlock**
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol
  - Shorter waiting times, and increase in concurrency
  - Protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee **recoverable** or **cascadeless** schedules
    - 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

Deadlock Handling

- A **deadlock** occurs if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

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)	

Deadlock Handling (Cont.)

- ***Deadlock prevention*** protocols ensure that the system does not enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
  - 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

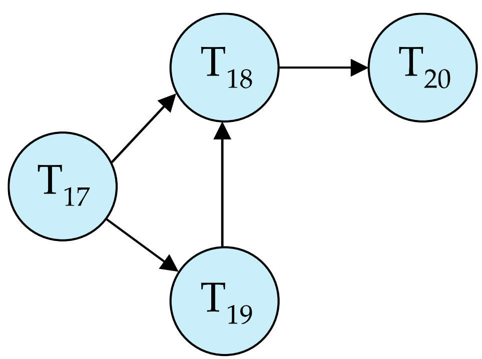
- **Wait-die** scheme
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **Wound-wait** scheme
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

Deadlock prevention (Cont.)

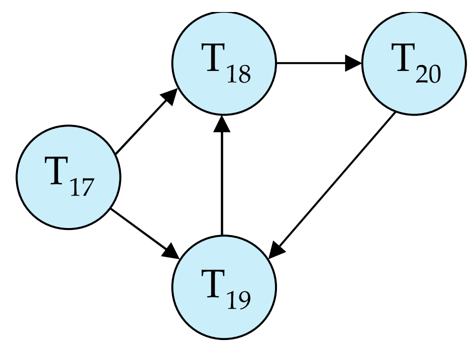
- **Timeout-based schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval.
  - Starvation is also possible

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from $T_i \rightarrow T_j$.* : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

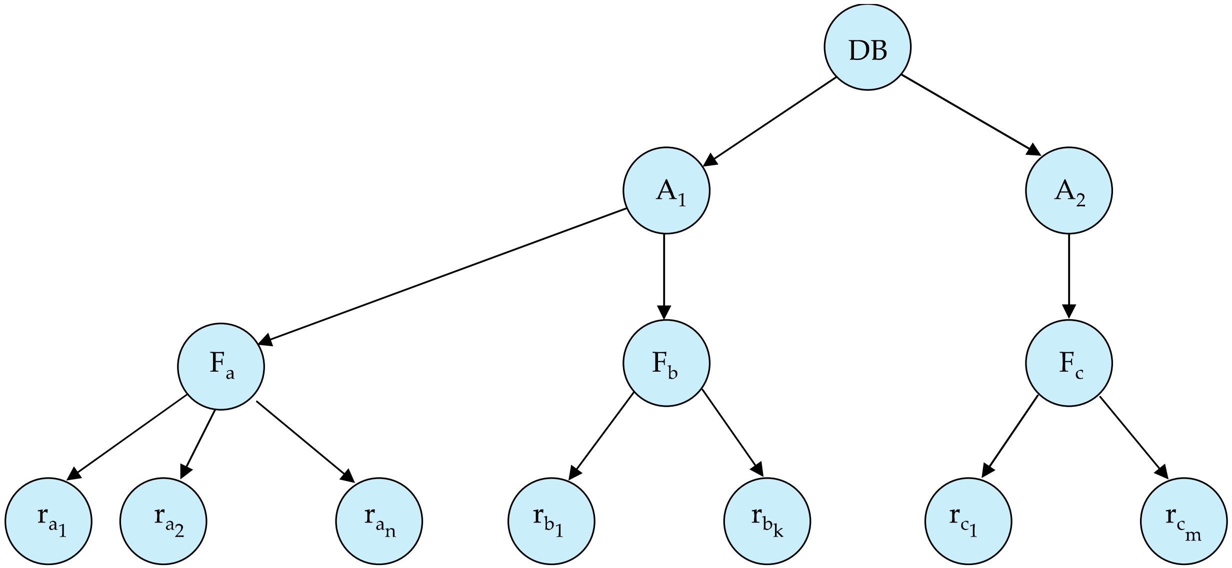
- When deadlock is detected:
  - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback – determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen
  - One solution: oldest transaction in the deadlock set is never chosen as victim

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
- The hierarchy can be represented graphically as a tree (but don't confuse with tree-protocol)
- 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 of Granularity Hierarchy

- The levels, starting from the coarsest (top) level can be
  - *database, area, file, record* (as in the book)
  - *database, table, page, row* (as in SQL Server)
  - etc.



- When a transaction locks a node in S or X mode, it *implicitly* locks all descendants in the same mode (S or X).

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates there are shared locks at lower levels of the tree
  - ***intention-exclusive*** (IX): indicates there are exclusive or shared locks at lowers level of the tree
  - ***shared and intention-exclusive*** (SIX): a shared lock, with the possibility of having exclusive or shared locks at lower levels of the tree.*
- With intention locks, a transaction does not need to search the entire tree to determine whether it can lock a node.

* The SIX mode was introduced as a refinement for transactions that read an entire subtree but update only a few nodes.

Multiple Granularity Locking Scheme

- A transaction can lock nodes according to the following rules:
  - The root of the tree is locked first in some mode (IS, IX, S, SIX, X).
  - If a node is locked in IS mode, its descendants can be locked in IS or S mode.
  - If a node is locked in IX mode, its descendants can be locked in any mode.
  - If a node is locked in S mode, its descendants are implicitly locked in S mode.
  - If a node is locked in SIX mode, its descendants are implicitly locked in S mode, but can also be locked IX, SIX, or X mode.
  - If a node is locked in X mode, its descendants are implicitly locked in X mode.

Multiple Granularity Locking Scheme (Cont.)

- In other words:
  - Before requesting an IS or S lock on a node, all ancestor nodes must be locked in IS or IX mode.
  - Before requesting an IX, SIX or X lock on a node, all ancestor nodes must be locked in IX or SIX mode.
- Leaf nodes are always locked in S or X mode
  - There are no intention locks on leaves since they have no descendants.

Multiple Granularity Locking Scheme (Cont.)

- Locks are acquired
  - in root-to-leaf order
- Locks are released
  - during the transaction, in leaf-to-root order
  - at the end of the transaction, in any order
- Re-acquiring locks after they have been released is not allowed

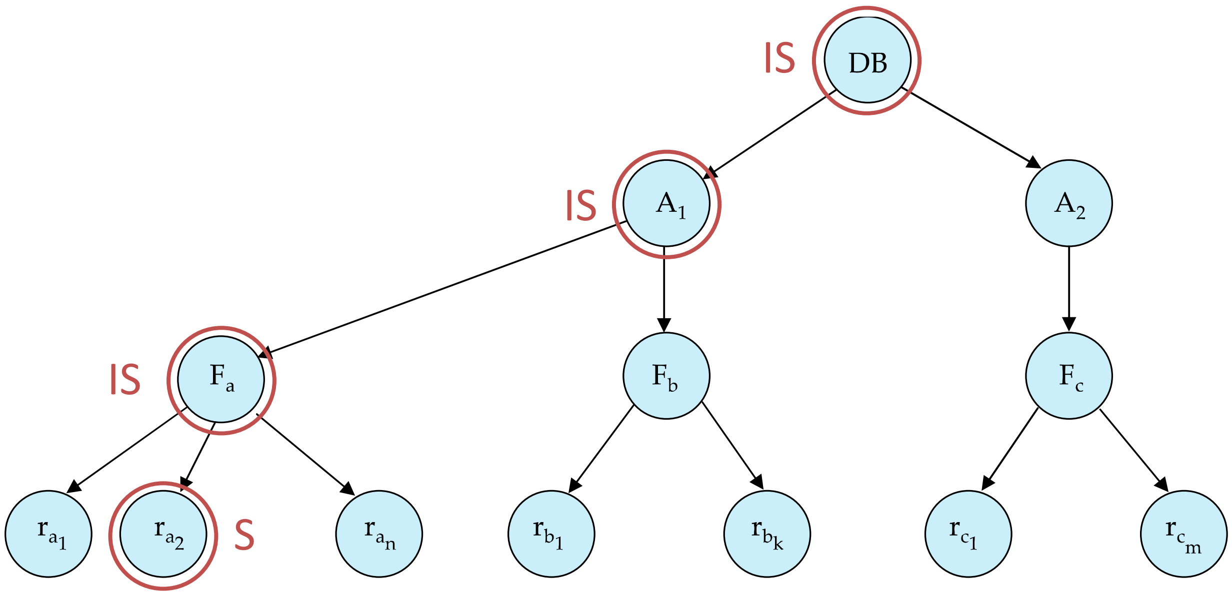
Compatibility Matrix with Intention Lock Modes

- The procedure is the same for all concurrent transactions
  - Locks will be granted according to the following compatibility matrix

	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

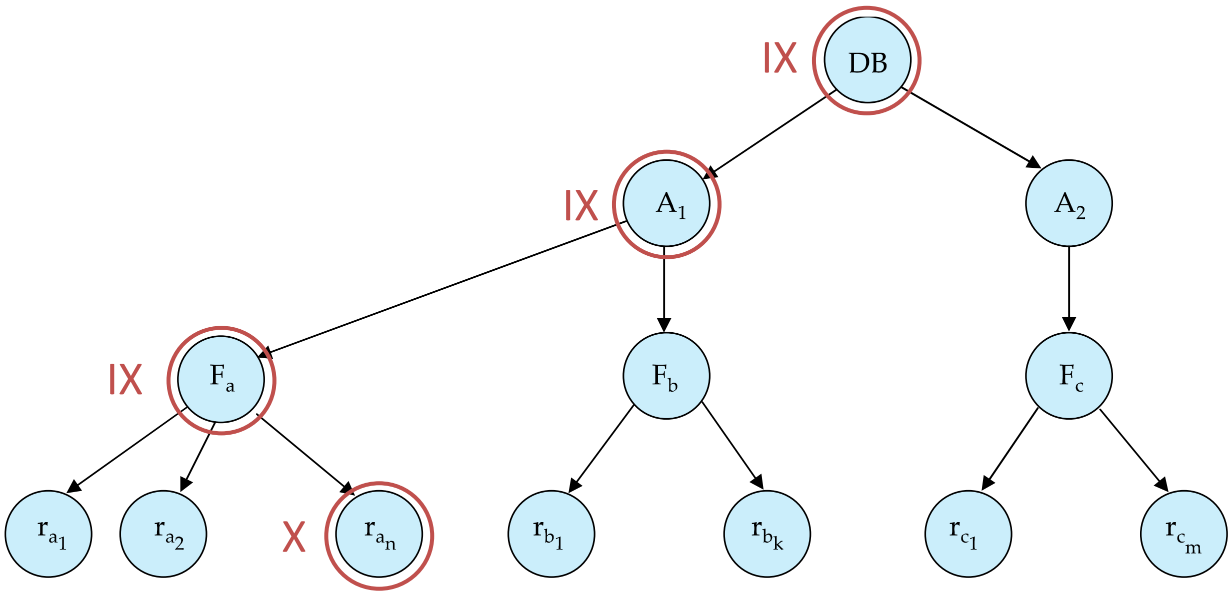
Multiple Granularity Locking Scheme: Example

- T_1: **read**(r_{a_2})



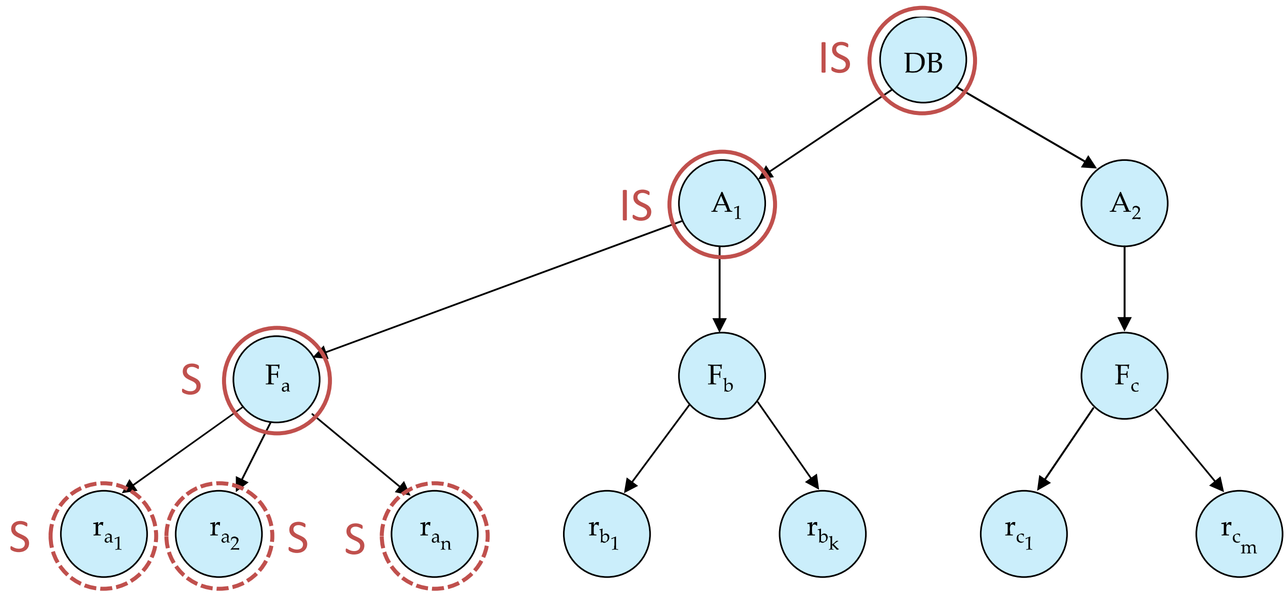
Multiple Granularity Locking Scheme: Example

- T_2: **write**(r_{a_9})



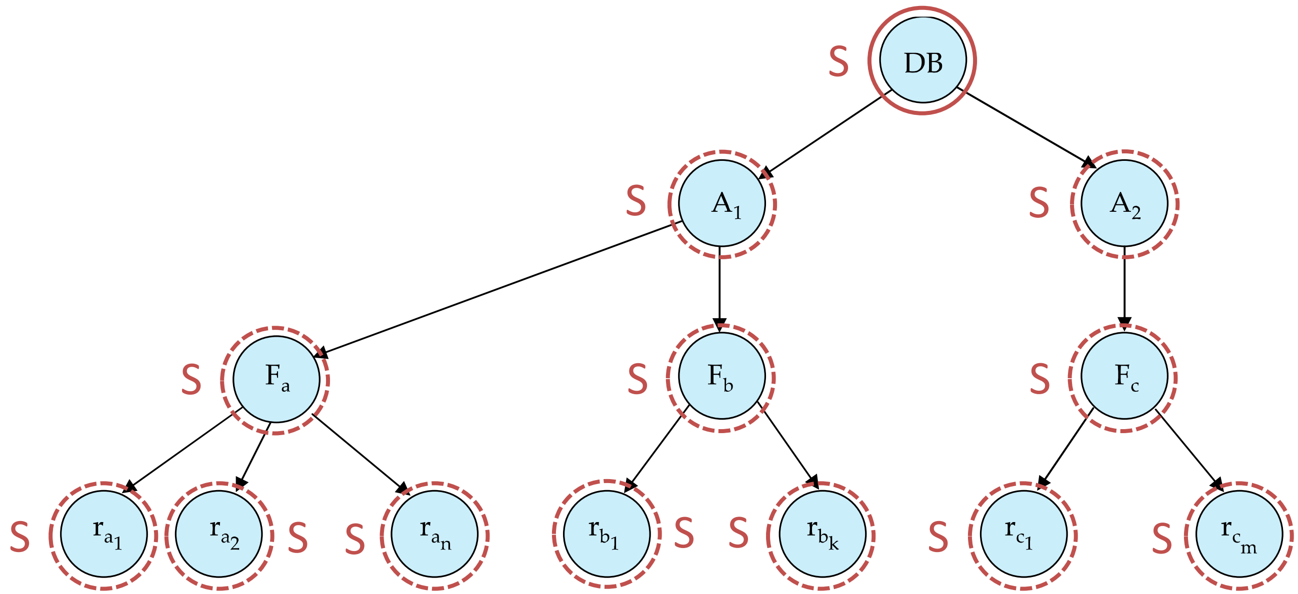
Multiple Granularity Locking Scheme: Example

- T_3: **read**(F_a)



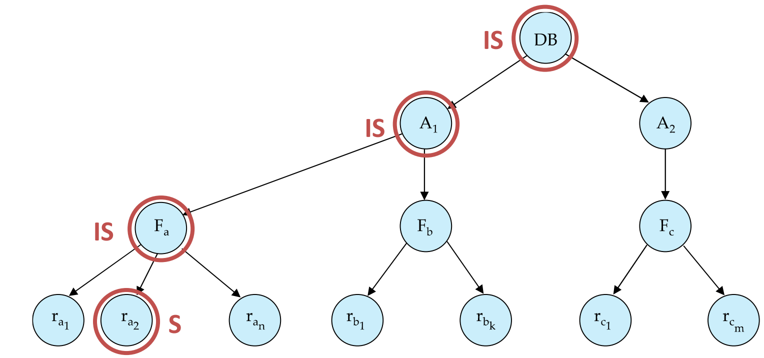
Multiple Granularity Locking Scheme: Example

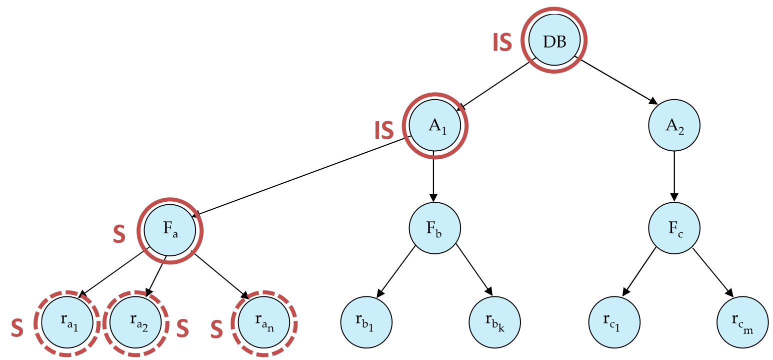
- T_4: **read**(DB)

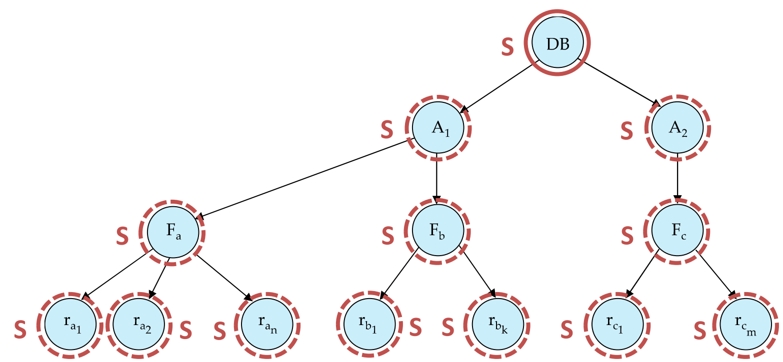


Multiple Granularity Locking Scheme: Example

- These are compatible:
  - T_1: **read**(r_{a_2})
  - T_3: **read**(F_a)
  - T_4: **read**(DB)



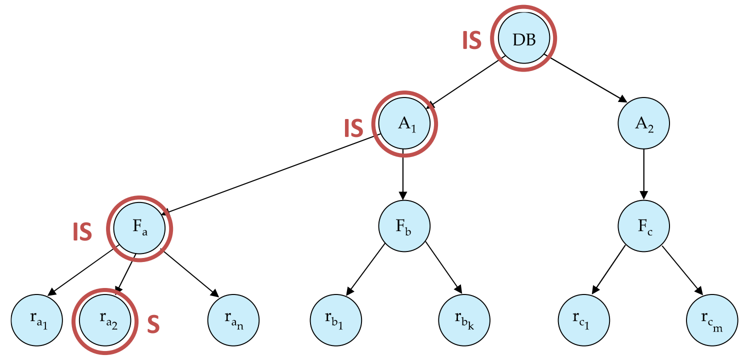


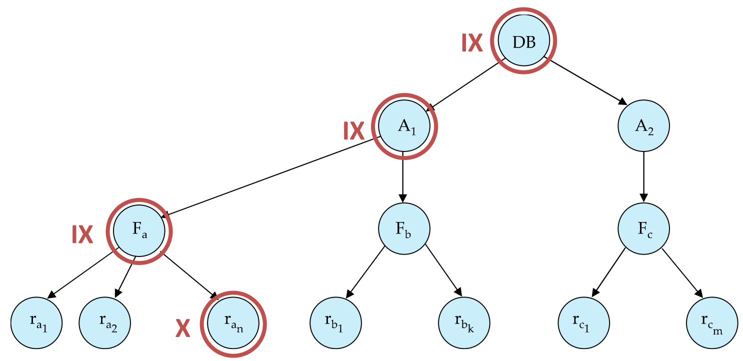


	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme: Example

- These are compatible:
  - T_1: **read**(r_{a_2})
  - T_2: **write**(r_{a_9})

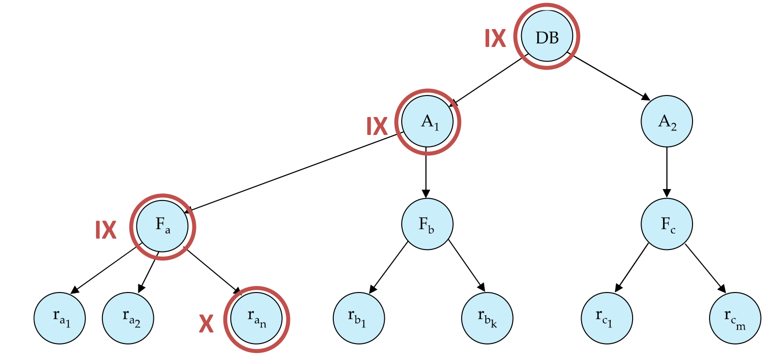


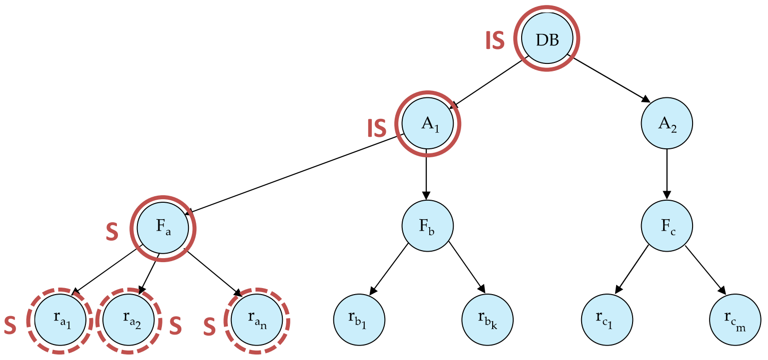


	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme: Example

- These are not compatible:
  - T_2: **write**(r_{a_9})
  - T_3: **read**(F_a)

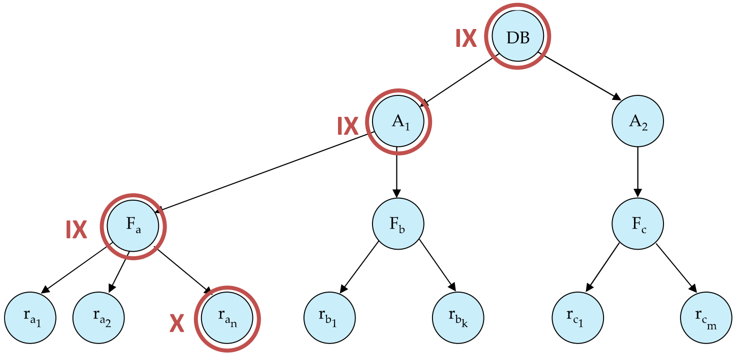


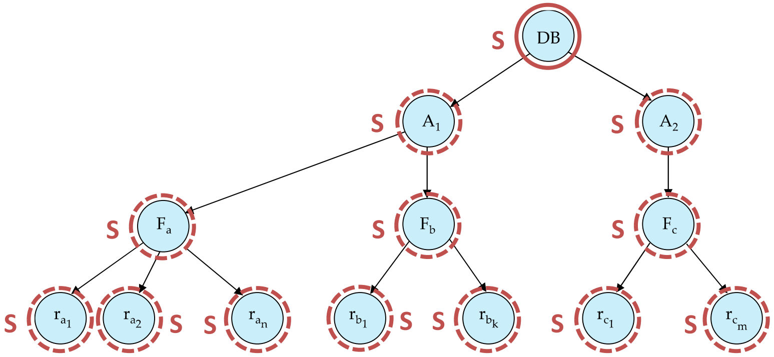


	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme: Example

- These are not compatible:
  - T_2: **write**(r_{a_9})
  - T_4: **read**(DB)





	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

Timestamp-Based Protocols

- Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps greater than earlier ones
  - Timestamp can be based on wall-clock time or logical counter
- Timestamp-based protocols manage concurrent execution such that **timestamp order = serializability order**
- Several protocols based on timestamps

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) protocol**

- Maintains for each data *Q* two timestamp values:
  - **W-timestamp**(*Q*) is the largest timestamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest timestamp of any transaction that executed **read**(*Q*) successfully.
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback

Timestamp-Ordering Protocol (Cont.)

- Suppose a transaction T_i issues a **read**(Q)

  1. If **W-timestamp**(Q) > TS(T_i), 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 **W-timestamp**(Q) ≤ TS(T_i), then the **read** operation is executed, and **R-timestamp**(Q) is set to

     max(**R-timestamp**(Q), TS(T_i)).

Timestamp-Ordering Protocol (Cont.)

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

  1. If **R-timestamp**(Q) > TS(T_i), then the value of Q that T_i is producing is being written too late, it should have been written earlier.
     - Hence, the **write** operation is rejected, and T_i is rolled back.

  2. If **W-timestamp**(Q) > TS(T_i), then T_i is attempting to write an obsolete value of Q; a newer transaction has written a more recent value.
     - 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).

Example of Schedule Under TSO

- This schedule is valid under TSO
  - Assume that initially:
    - **R-timestamp**(A) = **W-timestamp**(A) = 0
    - **R-timestamp**(B) = **W-timestamp**(B) = 0
  - Assume TS(T_{25}) = 25 and TS(T_{26}) = 26

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($A + B$)

Example of Schedule Under TSO (Cont.)

- This schedule is not valid under TSO
  - Assume that initially:
    - **R-timestamp**(Q) = **W-timestamp**(Q) = 0
  - Assume TS(T_{27}) = 27 and TS(T_{28}) = 28

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

  - T_{27} is attempting to write an obsolete value, and is therefore rolled back.

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
  - When T_i attempts to write data item Q, if **W-timestamp**(Q) > TS(T_i), 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.
  - Allows some schedules that are not conflict-serializable.

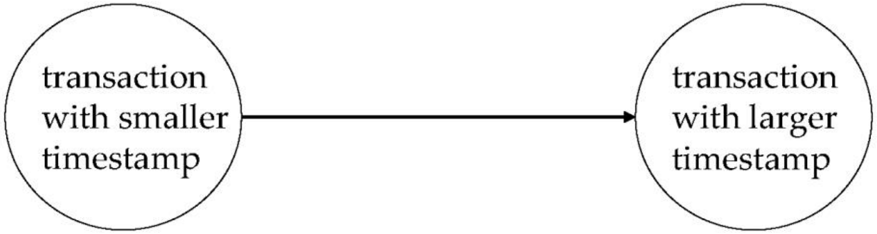
Another Example Under TSO

- A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5, with all **R-timestamp** = **W-timestamp** = 0 initially

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 prevents **deadlock** since no transaction ever waits.
- But the schedule may not be **cascade-free**, and may not even be **recoverable**.

Recoverability and Cascade Freedom

- 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

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Snapshot isolation**
- Key ideas:
  - 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 issuing the read request, and return the value of the selected version.
- Read requests never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$.
- Each version Q_k has its own timestamps:
  - **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

Multiversion Timestamp Ordering (Cont.)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let Q_k denote the version with the largest **W-timestamp** ≤ TS(T_i).
  1. If transaction T_i issues a **read**(Q), then
     - the value returned is version Q_k
     - If **R-timestamp**($\mathrm{Q_k}$) < TS(T_i), set **R-timestamp**($\mathrm{Q_k}$) = TS(T_i)
  2. If transaction T_i issues a **write**(Q)
     1. if **R-timestamp**(Q_k) > TS(T_i), then transaction T_i is rolled back.
     2. if **W-timestamp**(Q_k) = TS(T_i), then version Q_k is overwritten.
     3. Otherwise, a new version Q_i of Q is created, with **W-timestamp**(Q_i) = **R-timestamp**(Q_i) = TS(T_i)

Multiversion Timestamp Ordering (Cont.)

- Observations
  - Read requests never fail and never wait.
  - A write by T_i is rejected if some newer transaction T_j that should read T_i's version, has read a version created by a transaction older than T_i.
- Protocol guarantees **serializability**
  - but does not ensure **recoverability** or **cascadelessness**

Snapshot Isolation

- Widely used in practice (incl. Oracle, PostgreSQL, SQL Server, etc.)
- Each transaction is given its own snapshot of the database
  - Snapshot contains only committed values by previous transactions
  - Reads and writes are performed on the snapshot
  - Complete isolation between snapshots/transactions (before commit)
- Transactions that update the database have potential conflicts
  - Updates are kept in the snapshot until the transaction commits
  - Updates must be validated before the transaction is allowed to commit
  - If allowed to commit, updates in the snapshot are written to database
  - If not allowed to commit, transaction is rolled back
- Read requests never wait
  - Read from private snapshot
- Read-only transactions never fail
  - No updates, allowed to commit

Snapshot Isolation: Example

- A transaction T_i 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 T_i
  - Writes of T_i complete when it commits

T_1	T_2	T_3
write(Y) : 1 commit		
	start read(X) : 0 read(Y) : 1	
		write(X) : 2 write(Z) : 3 commit
	read(Z) : 0 read(Y) : 1 write(X) : 4 commit-req rollback	

Multiversioning in Snapshot Isolation

- In snapshot isolation, transactions are given two timestamps:
  - *StartTS*(T_i) is the time at which T_i started
  - *CommitTS*(T_i) is the time at which T_i requested commit
- Data items have versions, each with a single timestamp:
  - **W-timestamp**(Q_k) which is equal to *CommitTS*(T_i) of the transaction T_i that created version Q_k
- When a transaction T_j reads a data item Q
  - It reads the latest version Q_k such that **W-timestamp**(Q_k) ≤ *StartTS*(T_j)
  - It does not see any updates of transactions committed after *StartTS*(T_j)
  - T_j sees a snapshot of the database at the time when it started

Validation Steps in Snapshot Isolation

- Transactions T_i and T_j are said to be **concurrent** if either:
  - $StartTS(T_i) \leq StartTS(T_j) \leq CommitTS(T_i)$ or
  - $StartTS(T_j) \leq StartTS(T_i) \leq CommitTS(T_j)$
- When two concurrent transactions update the same data item
  - The two transactions operate in isolation in their own private snapshot
  - Neither transaction sees the update made by the other
  - If both transactions are allowed to commit and write to the database
    - one update will be overwritten by the other: **lost update**
- Two approaches to prevent lost updates:
  - **First committer wins**
  - **First updater wins**

Validation Steps in Snapshot Isolation (Cont.)

- **First committer wins**
  - T_i requests commit and is assigned *CommitTS*(T_i)
  - Suppose T_i has updated a single data item Q
  - If there is a version Q_k with *StartTS*(T_i) < **W-timestamp**(Q_k) < *CommitTS*(T_i)
    - A concurrent transaction has already written Q
    - T_i is not allowed commit, and must be rolled back
  - If no such version Q_k exists
    - T_i is allowed to commit, and its update is written to the database
  - Can be generalized to multiple data items (check all of them)

Validation Steps in Snapshot Isolation (Cont.)

- **First updater wins**
  - When T_i attempts to update data item Q, *it* requests a *write lock* on Q
  - If the lock is acquired:
    - If Q has been updated by a concurrent transaction, T_i is rolled back
    - Otherwise, T_i may proceed, while keeping the *write lock* on Q
  - If the lock is being held by a concurrent transaction T_j
    - T_i waits until T_j commits or aborts
    - If T_j aborts, T_i acquires the lock, and do the same as before
    - If T_j commits, T_i must be rolled back
  - The *write lock* on Q is released when T_i commits or aborts

Serializability in Snapshot Isolation

- Snapshot isolation does not ensure **serializability**
  - T_i reads A and B, updates A based on B
  - T_j reads A and B, updates B based on A
  - Updates are on different objects; both are allowed to commit
    - but the result is not equivalent to a serial schedule
  - Schedule is not conflict-serializable
    - Precedence graph has a cycle
  - This anomaly is called a **write skew**

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Serializable Snapshot Isolation

- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
  - For example, when T_i writes data item Q, and T_j reads an earlier version of Q, but T_j should be serialized after T_i
- **Serializable snapshot isolation (SSI)** is an extension of snapshot isolation that ensures serializability
  - Tracks both write-write and read-write conflicts
  - In theory, a transaction should be rolled back when a cycle is found
  - In practice, a transaction is rolled back when it has both an incoming read-write conflict and an outgoing read-write conflict
    - may result in some unnecessary rollbacks, but it's simpler to check