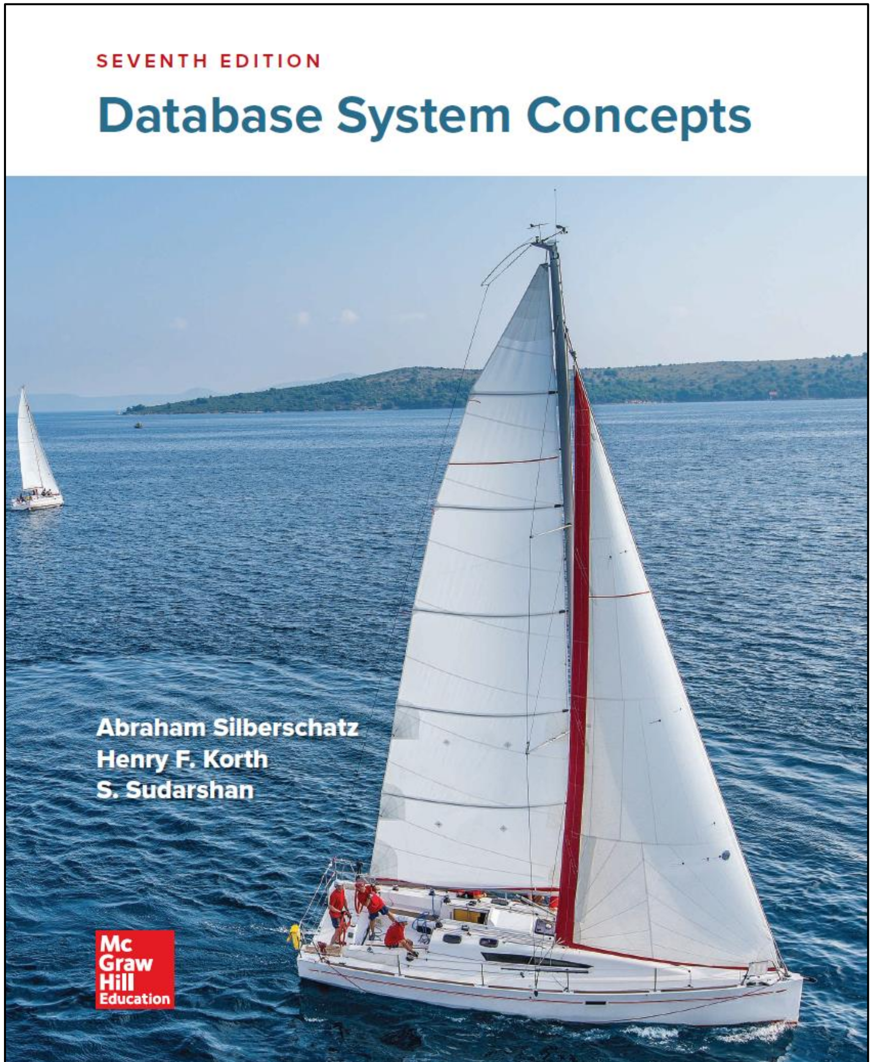
Data Administration in Information Systems

Transactions and concurrency

Query optimization



Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer 50€ from account *A* to account *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

  Steps 1–3: **update** *account* **set** *balance* = *balance* – 50 **where** *account_number* = *x*

  Steps 4–6: **update** *account* **set** *balance* = *balance* + 50 **where** *account_number* = *y*
- Two main issues to deal with:
  - Concurrent execution of multiple transactions
  - Failures of various kinds, such as hardware failures and system crashes

Example of Fund Transfer

- Transaction to transfer 50€ from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the 50€ has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of *A* and *B* is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database
  - During transaction execution the database may be temporarily inconsistent
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1	**T2**
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), **read**(B), **print**(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*)	

- Isolation can be ensured trivially by running transactions **serially**
  - i.e. one after the other
- However, executing multiple transactions concurrently has significant benefits

ACID Properties

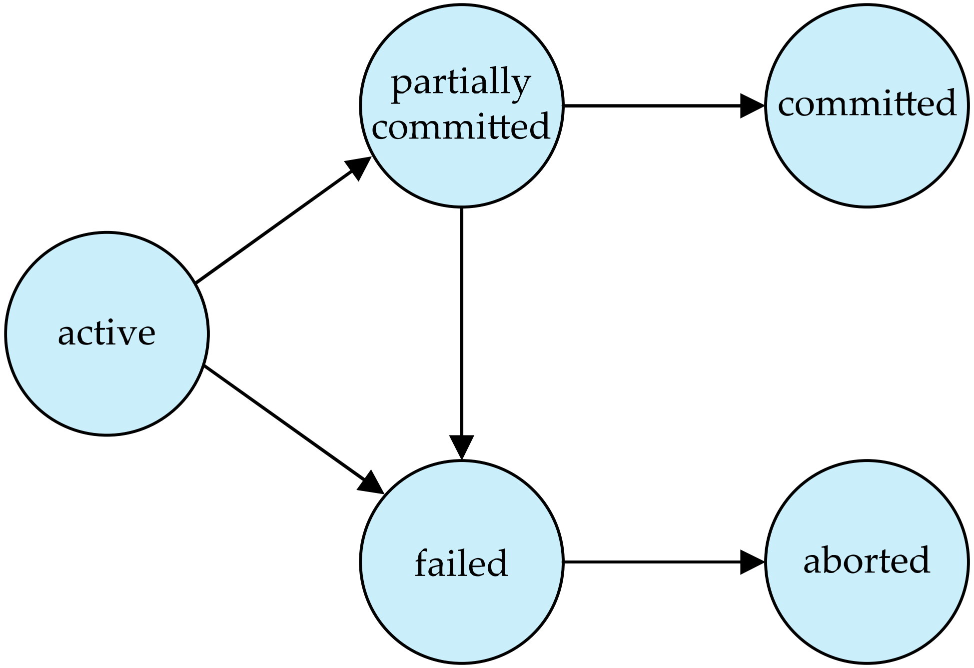
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - i.e. to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instruction as the last statement
  - By default, a transaction is assumed to execute a commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer 50 € from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- A serial schedule in which T_1 is followed by T_2:

T_1	T_2
read (*A*) *A* := *A* – 50 write (*A*) read (*B*) *B* := *B* + 50 write (*B*) commit	
	read (*A*) *temp* := *A* * 0.1 *A* := *A* - *temp* write (*A*) read (*B*) *B* := *B* + *temp* write (*B*) commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A)
	$temp$:= A * 0.1
	A := A - $temp$
	write (A)
	read (B)
	B := B + $temp$
	write (B)
	commit
read (A)	
A := A – 50	
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

T_1	T_2
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit

- In Schedules 1, 2 and 3, the sum $A + B$ is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of $A + B$

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A concurrent schedule is serializable if it is equivalent to a serial schedule.
- We focus on a particular form of schedule equivalence called **conflict serializability**

Conflicting Instructions

- There is a **conflict** between transactions T_i and T_j if and only if there exists some item Q accessed by both transactions, and at least one of them writes Q.
    1. T_i : **read**(Q) T_j : **read**(Q) No conflict
    2. T_i : **read**(Q) T_j : **write**(Q) Conflict
    3. T_i : **write**(Q) T_j : **read**(Q) Conflict
    4. T_i : **write**(Q) T_j : **write**(Q) Conflict
- Intuitively, a conflict between T_i and T_j forces a (logical) temporal order between them.
- If the instructions of T_i and T_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore, Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

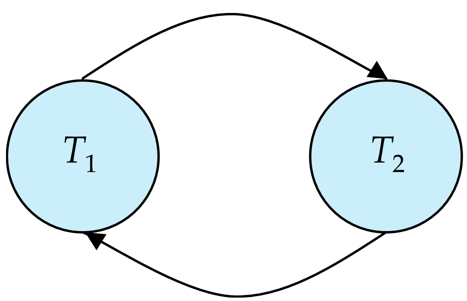
- Example of a schedule that is not conflict serializable:

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

  - We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3 , T_4 >, or the serial schedule < T_4 , T_3 >.

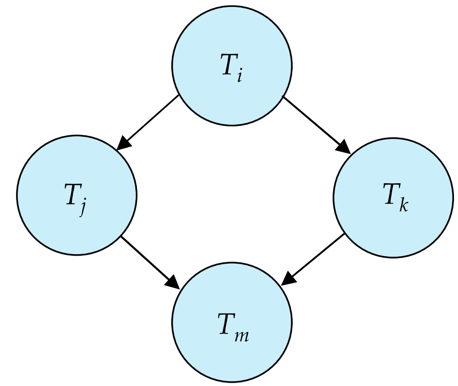
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example of a precedence graph

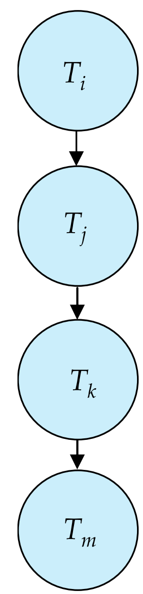


Test for Conflict Serializability

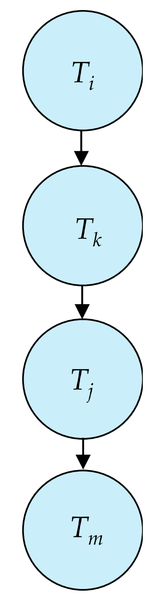
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- If precedence graph is acyclic, the serializability order can be obtained by a *linear sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for schedule (a) could be (b) or (c).



(a)



(b)



(c)

Test for Conflict Serializability: Examples

- The precedence graph for this schedule does not contain cycles
  - It is conflict serializable

T_1	T_2
read (*A*) *A* := *A* – 50 write (*A*)	
	read (*A*) *temp* := *A* * 0.1 *A* := *A* - *temp* write (*A*)
read (*B*) *B* := *B* + 50 write (*B*) commit	
	read (*B*) *B* := *B* + *temp* write (*B*) commit

Test for Conflict Serializability: Examples

- The precedence graph for this schedule contains a cycle
  - It is not conflict serializable

T_1	T_2
read (A) $A := A - 50$	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B)
write (A) read (B) $B := B + 50$ write (B) commit	
	$B := B + temp$ write (B) commit

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule < T_1 , T_5 >, yet is not conflict serializable.

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)

  – Determining such equivalence requires analysis of operations other than **read** and **write**.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if transaction T_j reads a data item previously written by a transaction T_i, then the **commit** of T_j must appear after the **commit** of T_i
- The following schedule is not recoverable:

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

  - If T_8 rolls back, T_9 has read an inconsistent database state.
  - Database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable):

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

  – If T_{10} fails, T_{11} and T_{12} must also be rolled back.
  – This can lead to the undoing of a significant amount of work.

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur.
  - If transaction T_j reads a data item previously written by a transaction T_i , then the **read** of T_j must appear after the **commit** of T_i.
- Every **cascadeless schedule** is also **recoverable**
  - Because if the **read** of T_j appears after the **commit** of T_i, then the **commit** of T_j will also appear after the **commit** of T_i.
- It is desirable to restrict the schedules to those that are **cascadeless**

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - **serializable**, and
  - **recoverable**, preferably **cascadeless**
- If only one transaction executes at a time, this generates serial schedules, but provides a poor degree of concurrency
  - Concurrency-control schemes allow concurrency while trying to comply with the requirements above.
- Testing a schedule for serializability after it has been executed is too late!
- **Goal** – develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency control protocols allow concurrent schedules, but ensure that the schedules are serializable, recoverable, and preferably cascadeless.
- Concurrency control protocols do not have access to the precedence graph until the transactions are finished.
  - Therefore, a protocol imposes a discipline that avoids non-serializable schedules (more about this later).
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - e.g. a read-only transaction that wants to get an approximate total balance of all accounts
  - e.g. database statistics computed for query optimization can be approximate
  - such transactions need not be serializable with respect to other transactions
- Tradeoff between accuracy and performance

Levels of Consistency in SQL

- **Serializable** — ensures serializable execution.
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable; it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of a record may return different (committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency in SQL (Cont.)

Isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Serializable	no	no	no
Repeatable read	no	no	yes
Read committed	no	yes	yes
Read uncommitted	yes	yes	yes

- **Dirty reads**: the transaction can see the changes being done by other running transactions which have not commited yet.
- **Non-repeatable read**: the data in a record may appear to change due to other transactions that have committed in the meantime.
- **Phantom reads**: the number of records in a table may appear to change due to other transactions that have committed in the meantime.

Levels of Consistency in SQL (Cont.)

- Lower degrees of consistency useful for gathering approximate information about the database
- Some systems do not ensure serializable schedules by default
  - Default isolation level is typically **read committed** or **repeatable read**
- Some systems have additional isolation levels
  - **Snapshot isolation** (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly
  - By default, each statement is a transaction that commits upon successful execution.
  - "Auto-commit" can be turned off, if desired.
- Explicit transactions start with **begin transaction** and end with **commit** or **rollback**
  - In most systems, the transaction is rolled back automatically upon error.
- The isolation level can be changed before the start of a new transaction
  - With the command **set transaction isolation level ...**

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)	

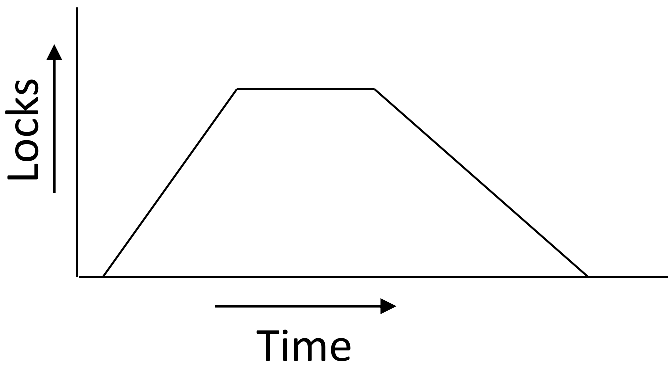
- Neither T_3 nor T_4 can make progress
  - executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To break the deadlock, one of T_3 or T_4 must be 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 Ti has a lock-X on D
  then
   write(D)
 else begin
    if needed, wait until no other transaction has any lock on D
    if Ti has a lock-S on D
       then
         upgrade lock on D to lock-X
      else
         grant Ti a lock-X on D
      write(D)
  end;
```

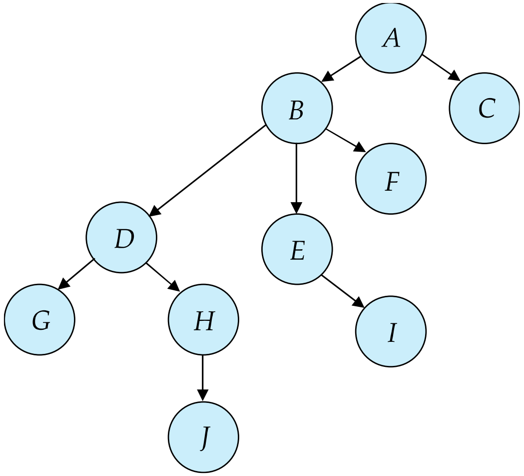
- All locks are released after commit or abort

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

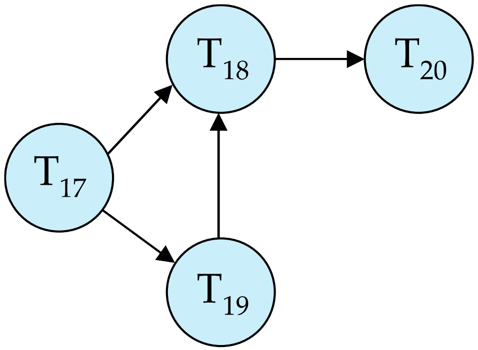
- **Wait-die** scheme
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **Wound-wait** scheme
  - Older transaction *wounds* (forces rollback) 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.)

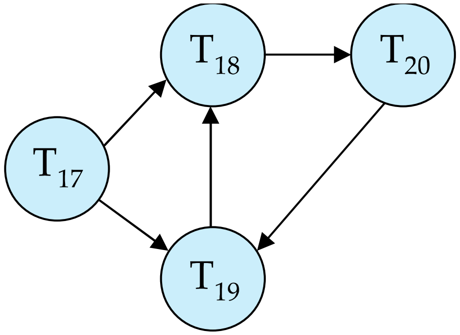
- **Timeout-based schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily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

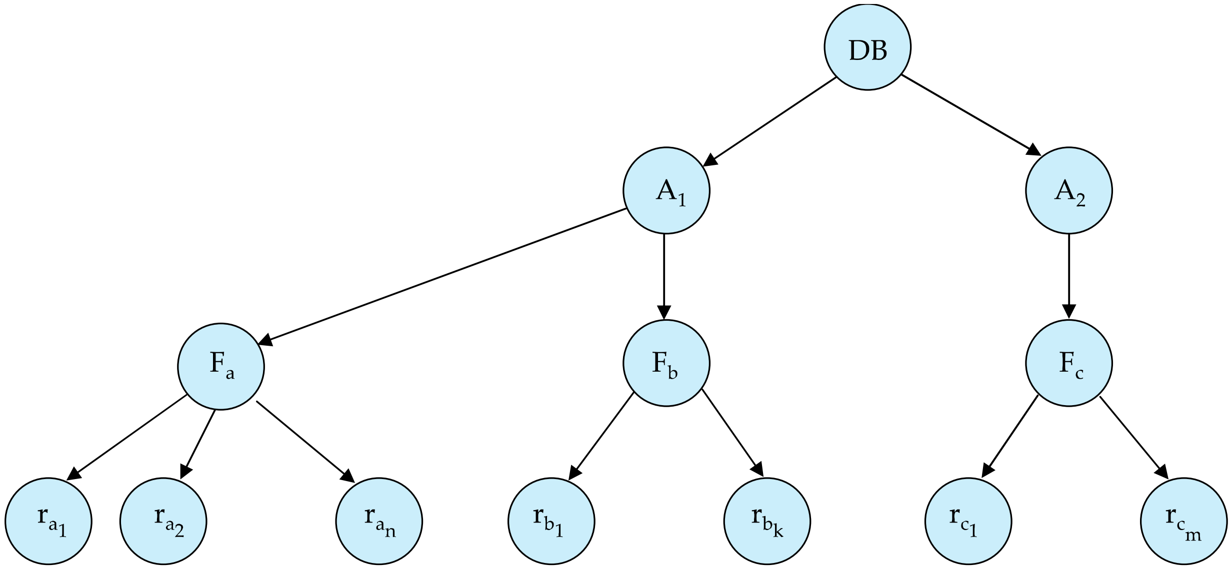
- When deadlock is detected:
  - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback – determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then 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.

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.

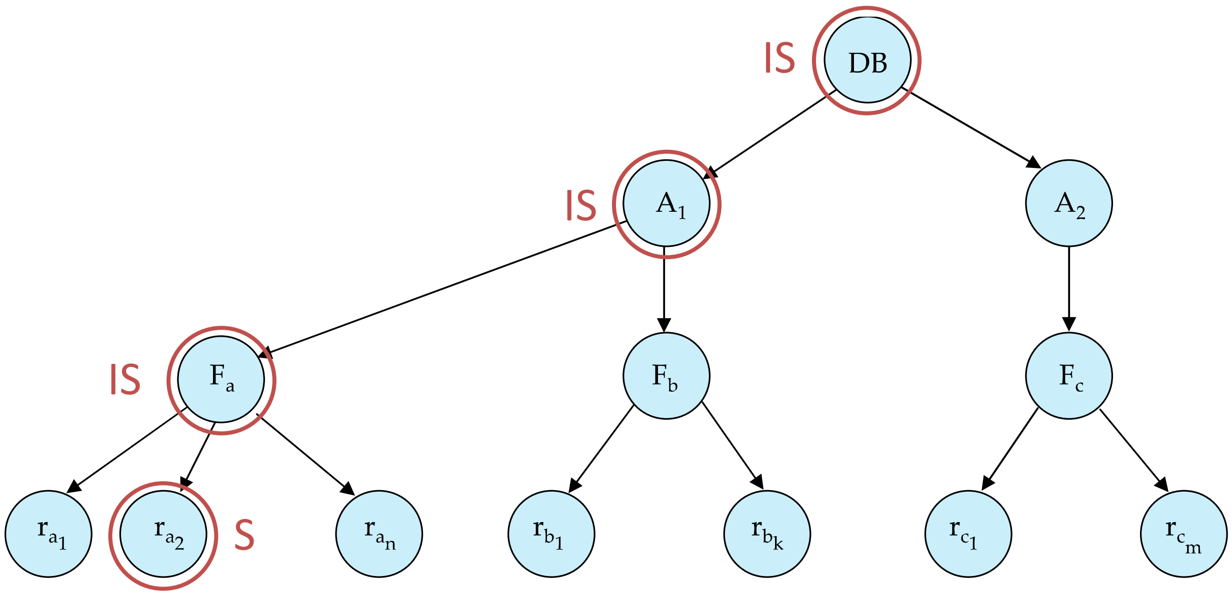
Compatibility Matrix with Intention Lock Modes

- 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

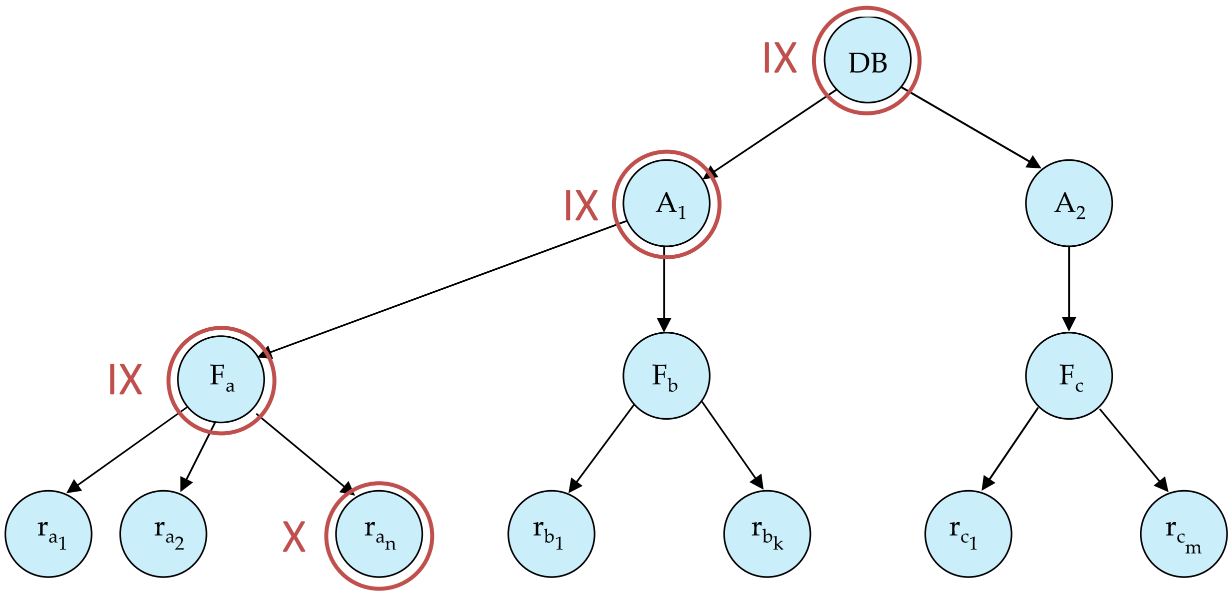
Multiple Granularity Locking Scheme: Example

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



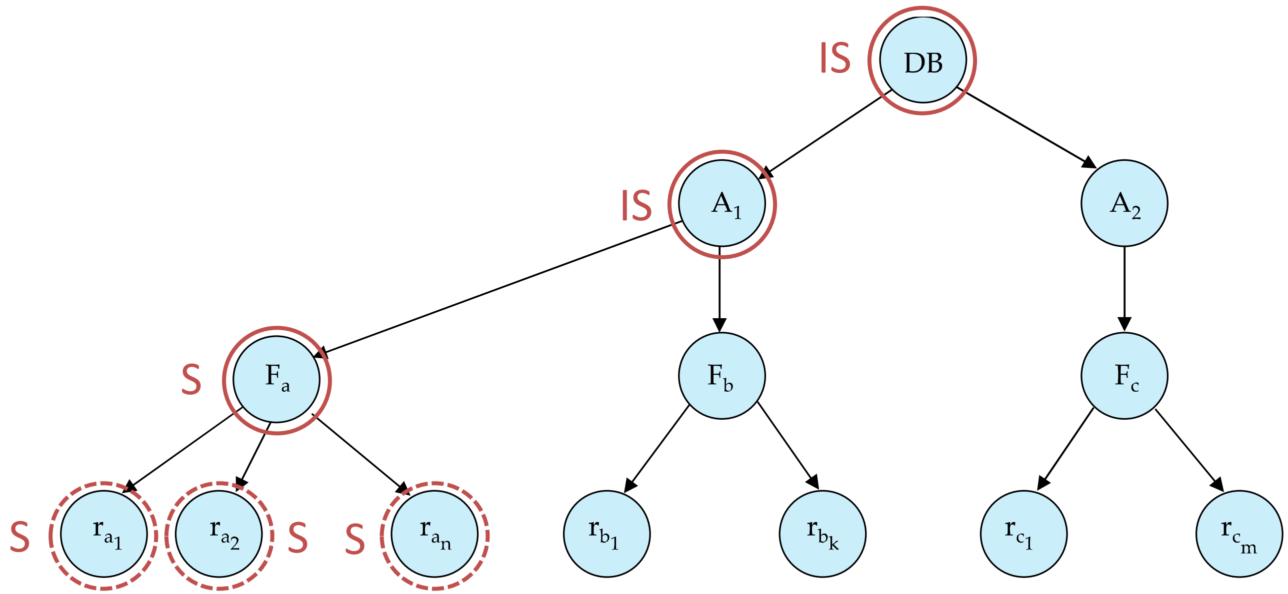
Multiple Granularity Locking Scheme: Example

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



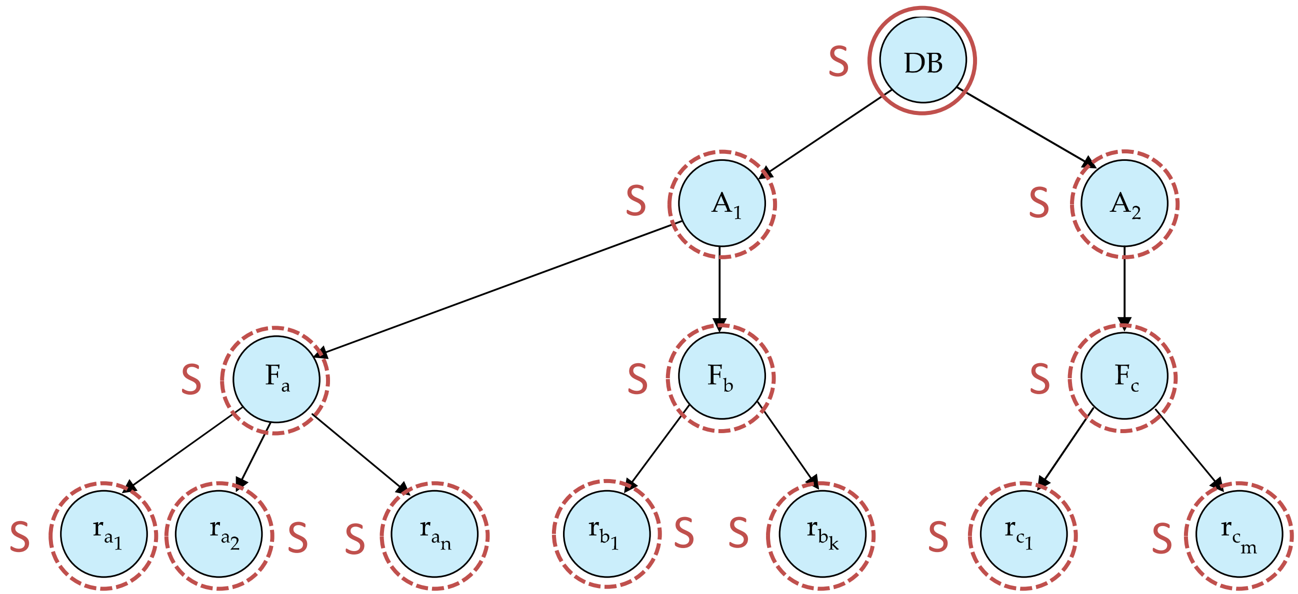
Multiple Granularity Locking Scheme: Example

- T_3: **read**(F_a)



Multiple Granularity Locking Scheme: Example

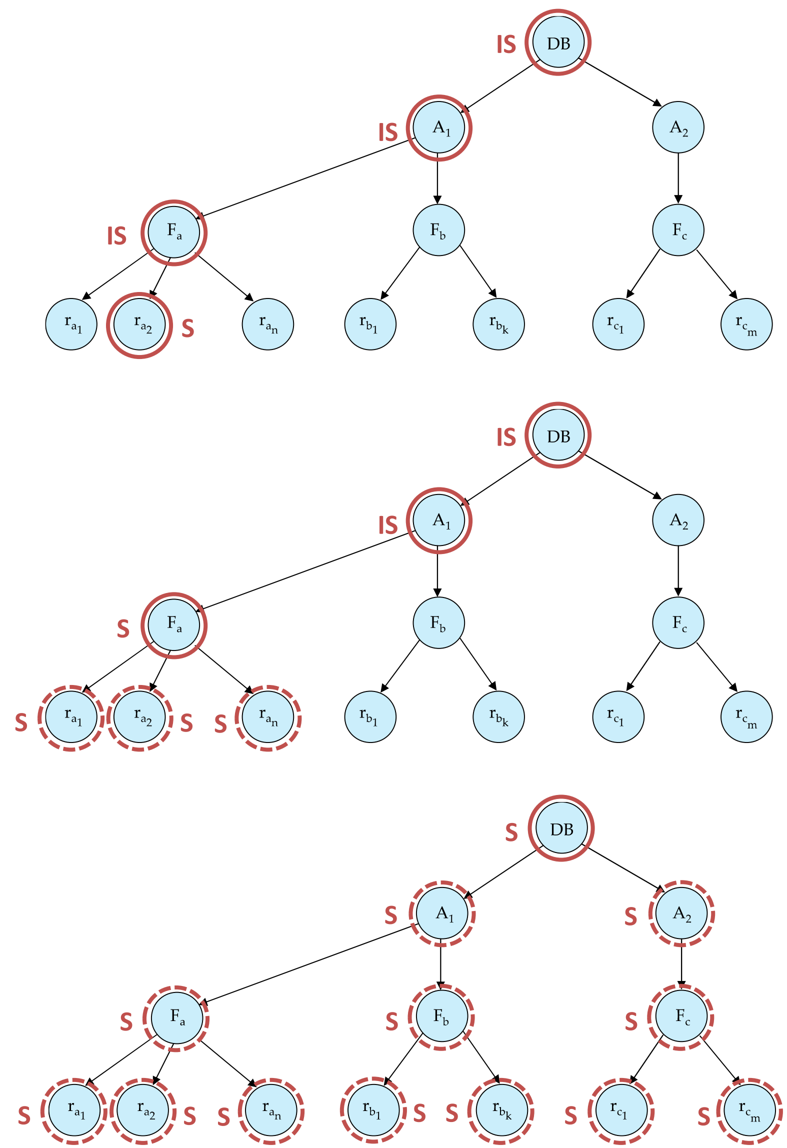
- T_4: **read**(DB)



Multiple Granularity Locking Scheme: Example

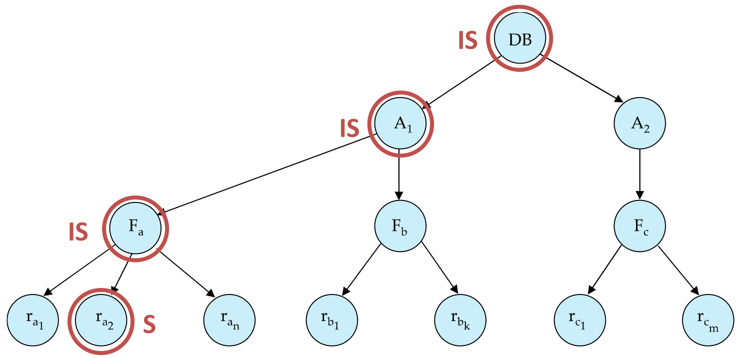
- These are compatible:
  - T_1: **read**(r_{a_2})
  - T_3: **read**(F_a)
  - T_4: **read**(DB)

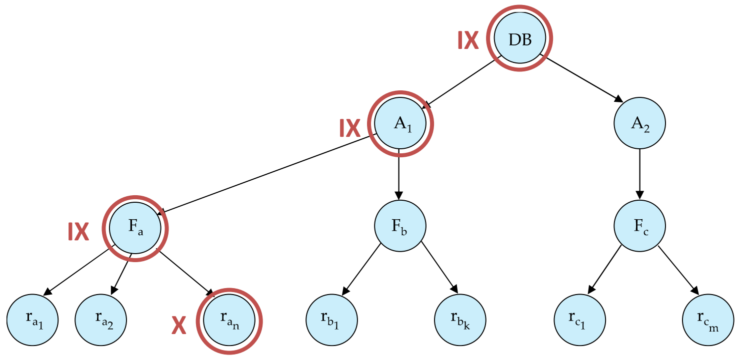
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false



Multiple Granularity Locking Scheme: Example

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

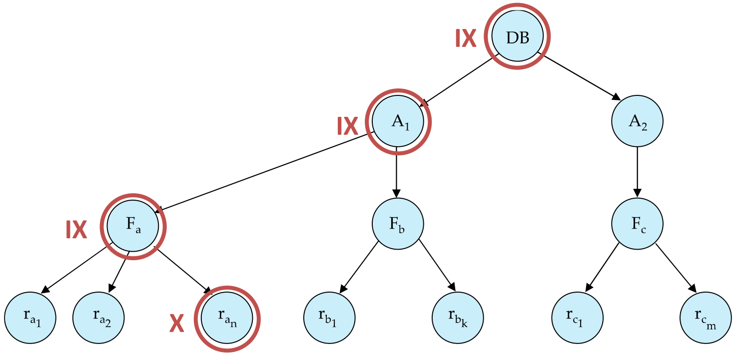


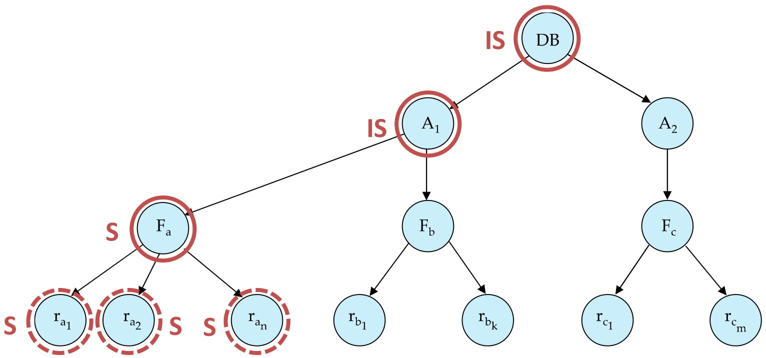


	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme: Example

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

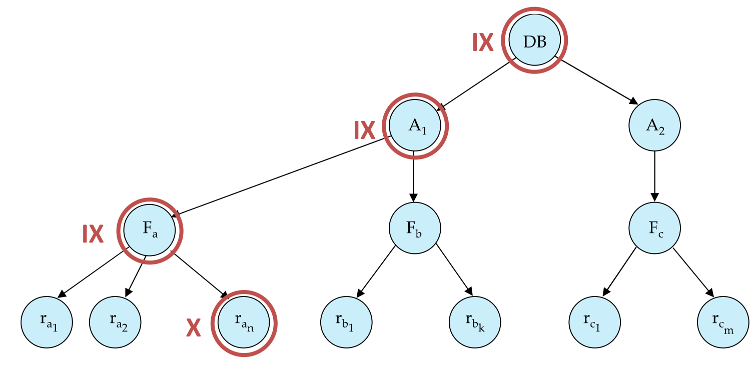


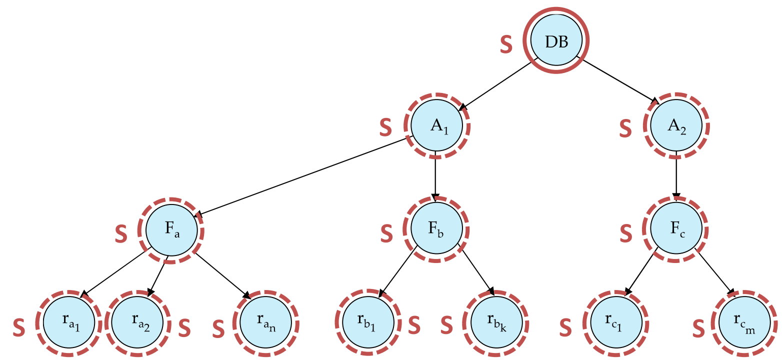


	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

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.

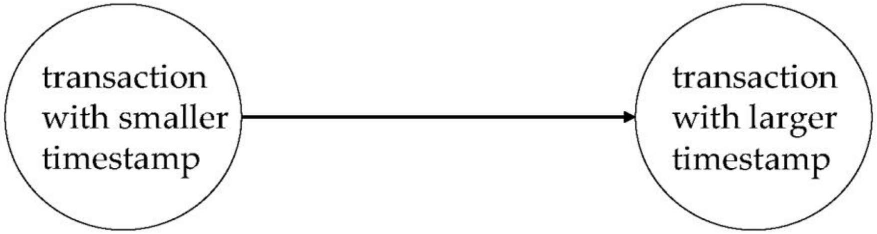
Another Example Under TSO

- 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,...., 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**(Q_k) < TS(T_i), set **R-timestamp**(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