COP 3540: Introduction to Database Structures

Fall 2017

Transaction Management

M. Rathod

Transaction

Database ‘objects’ are the units in which programs read or write information (units could be pages, records)

Database to be a *fixed* collection of *independent* objects

Transaction is a series of reads and writes of database objects

- **read**: object is brought into main memory (frame in the buffer pool) from disk, and its value is copied into a program variable
- **write**: an in-memory copy of the database object is first modified and then written to disk

Transaction

4 Important Transaction Properties (ACID):

1. **atomicity**: each transaction execution must carry out all or none of the actions. There can not be any incomplete transactions
2. **consistency**: each transaction runs by itself maintaining database consistency (responsibility of the user)
3. **isolation**: transactions must be isolated, or protected, from the effects of concurrently scheduling other transactions
4. **durability**: after the transaction completes successfully, the effects must persist despite a system crash and before all changes are written on disk

Atomicity and Durability

Transactions can be incomplete due to:

1. unsuccessful termination/aborted - because some anomaly arises during execution (can be restarted)
2. system crash (power supply)
3. unexpected situation (read an unexpected data value or be unable to access some disk)

- DBMS ensures transaction atomicity by undoing the actions of incomplete transactions
- A log writes completed transactions to disk when system restarts

Transactions and Schedules

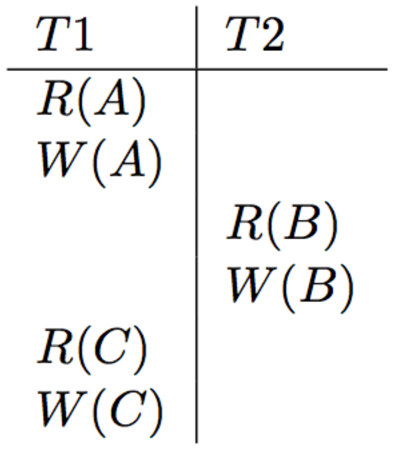
Transaction can also be defined as a set of actions that are partially ordered

- R_T (O) or R(O) - transaction T reading an object O
- W_T (O) or W(O) - transaction T writing an object O
- $Abort_T$ - terminate and undo all the actions carried out thus far
- $Commit_T$ - complete successfully

- **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions
- order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T

Transactions and Schedules

- a schedule represents an actual or potential execution sequence (as seen by the DBMS)

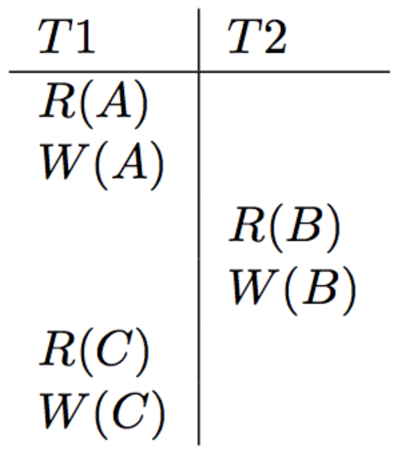


$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

- **complete schedule** – has an abort or a commit for each transaction whose actions are listed
- **serial schedule** - transactions are executed from start to finish, one by one (different transactions are not interleaved)

Concurrent Transaction Execution

- Why have concurrency?



$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

1. **system throughput** – overlapping I/O and CPU activity reduces the amount of time disks and processors are idle
2. **response time** - interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly

Concurrent Transaction Execution

- Each transaction must preserve database consistency
- **Consistent database state** is defined during design phase (e.g.: employee salaries must be <80% of department budget)
- When a complete serial schedule is executed against a consistent database, result is a consistent database
- **Serializability:** database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in some serial order

Concurrent Transaction Execution

Anomalies Associated with Interleaved Execution:

- two consistency preserving, committed transactions could run against a consistent database and leave it in an inconsistent state

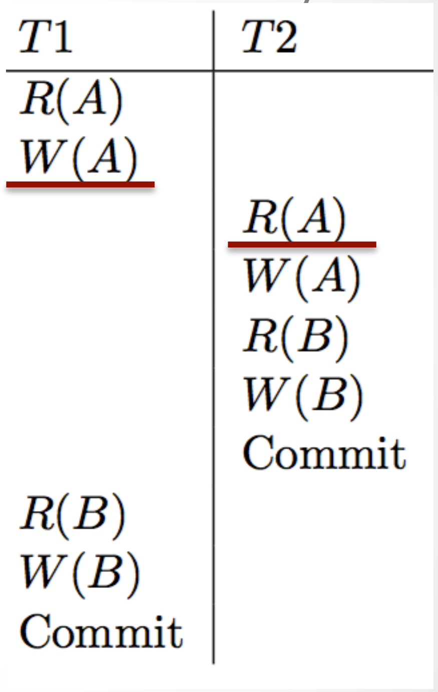
- Two actions on the same data object **conflict** if at least one of them is a write

three anomalous situations

- write-read (WR) conflicts
- read-write (RW) conflicts
- write-write (WW) conflicts

Write-Read (WR) Conflicts

dirty read: transaction T2 could read a database object A that has been modified by another transaction T1, which has not yet committed



$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

T1 transfers $100 from A to B

T2 increments both A and B by 6%

Actions interleaved:

(1transfer program T1 deducts $100 from A

(2)interest deposit program T2 reads current values of accounts A and B & adds 6% interest to each

(3) account transfer program credits $100 to account B

A written by T1 is read by T2 before T1 has completed all its changes! (reading uncommmitted data)

Read-Write (RW) Conflicts

Unrepeatable Reads (RW Conflicts)

T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress

1) **unrepeatable read**: when T1 tries to read the value of A again, it will get a different result, even though it has not modified A

2) both T1 and T2 read the same value of A (5), then T1 increments A by 1 (6) and T2 decrements A the value that it reads (5) by 1 and changes A to 4.

although T2's change is not directly read by T1, it invalidates T1's assumption about the value of A, which is the basis for some of T1's subsequent actions

Write-Write (WW) Conflicts

Overwriting Uncommitted Data (WW Conflicts)

T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress

Example: Two salaries of employee E1 and E2 must be kept equal

T1 sets their salaries to $1,000 and transaction T2 sets their salaries to $2,000

If we execute these in the serial order T1 followed by T2, both receive the salary $2,000 (or $1000)

neither transaction reads a salary value before writing it—such a write is called a **blind write**

Write-Write (WW) Conflicts

Overwriting Uncommitted Data (WW Conflicts)

T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress

Example: Two salaries of employee E1 and E2 must be kept equal. Interleaving of the actions of T1 and T2:

T1 sets E1's salary to $1,000, T2 sets E2's salary to $2,000

T1 sets E2's salary to $1,000, and finally T2 sets E1's salary to $2,000

result is not identical to the result of either of the two possible serial executions, and the interleaved schedule is therefore not serializable

Schedules - Aborted Transactions

all actions of aborted transactions are to be undone

serializable schedule over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of committed transactions in S

serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations

Schedules - Aborted Transactions

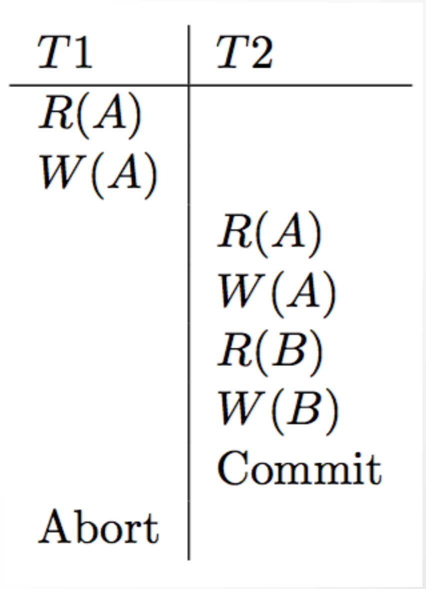
(1) account transfer program T1 deducts $100 from account A

(2) interest deposit program T2 reads the current values of accounts A and B and adds 6 percent interest to each, then commits

(3) T1 is aborted

T2 has read a value for A that should never have been there! schedule is unrecoverable

If T2 had not yet committed we could cascade the abort of T1 and abort T2 (recursively abort any transaction that read data written by T2).



$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
Abort	

Schedules - Aborted Transactions

recoverable schedule - transactions commit only after all transactions whose changes they read commit.

avoid cascading aborts - read only the changes of committed transaction and aborting a transaction can be accomplished without cascading the abort

A has value 5

T1 changes A o 6

T2 changes A to 7

T1 now aborts, A becomes 5 again

Even if T2 commits, its change to A is lost!

Lock-Based Concurrency Control

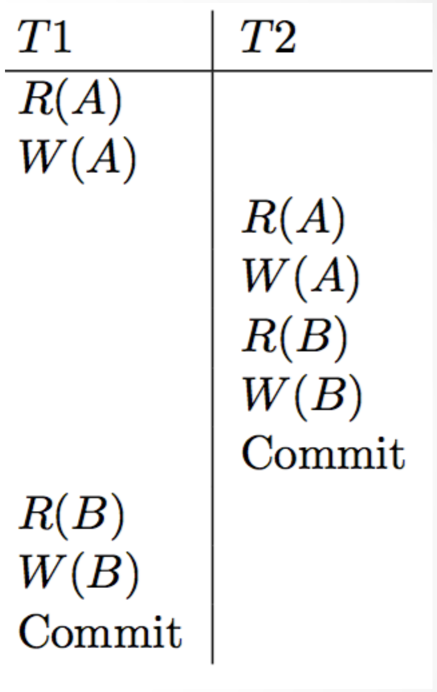
locking protocol is a set of rules to be followed by each transaction (enforced by the DBMS) - to ensure that even if actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Strict Two-Phase Locking (Strict 2PL)

Rule(1) If T wants to read (and modify) an object, it first requests a shared (and then exclusive) lock on the object

Rule(2) All locks held by a transaction are released when the transaction is completed

Lock-Based Concurrency Control



$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Example:

T1 changes A from 10 to 20

T2 reads 20 for A, and

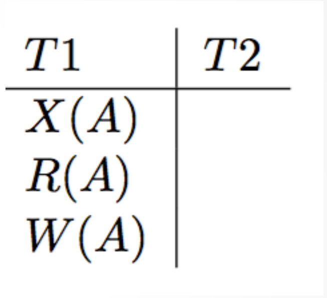
changes B from 100 to 200

T1 reads the value 200 for B

If run serially, either T1 or T2 would execute first, and read the values 10 for A and 100 for B - interleaved execution is not equivalent to either serial execution

Lock-Based Concurrency Control

Strict Two-Phase Locking (Strict 2PL)



$T1$	$T2$
$X(A)$	
$R(A)$	
$W(A)$	

Shared lock on object O - $S_T(O)$ or $S(O)$

Exclusive lock on O - $X_T(O)$ or $X(O)$

T1 would obtain an exclusive lock on A first and then read and write A

T2 would request a lock on A but request cannot be granted until T1 releases its exclusive lock on A

DBMS suspends T2

T1 obtains an exclusive lock on B, reads and writes B, commits, locks are released

T2's lock request is now granted, and it proceeds

Lock-Based Concurrency Control

Strict Two-Phase Locking (Strict 2PL)

Serial:

T1 obtains an exclusive lock on A

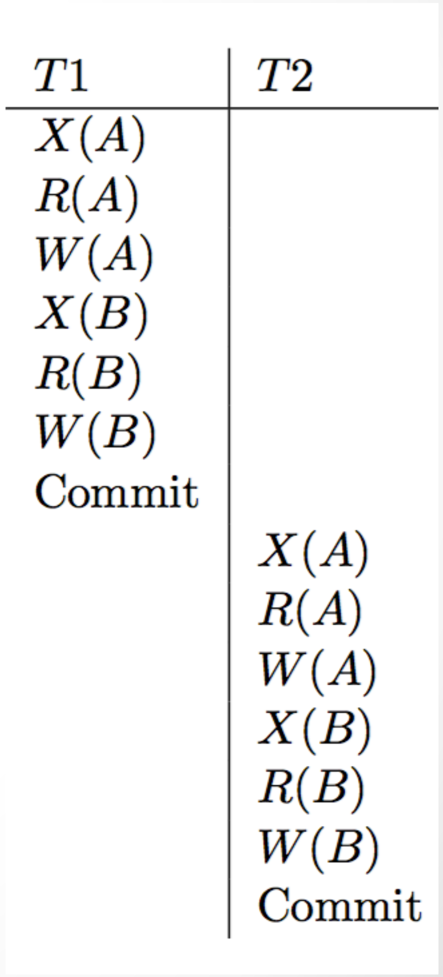
and reads and writes A

T1 obtains an exclusive lock on B

and reads and writes B

commits, locks are released

T2's lock request is now granted



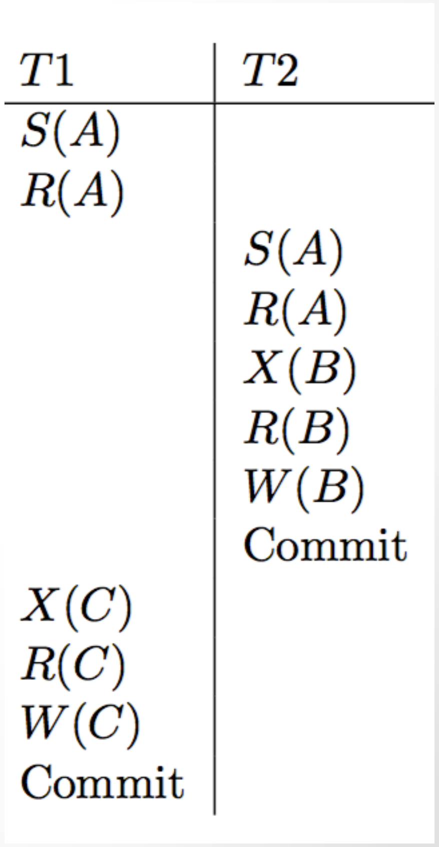
$T1$	$T2$
$X(A)$	
$R(A)$	
$W(A)$	
$X(B)$	
$R(B)$	
$W(B)$	
Commit	
	$X(A)$
	$R(A)$
	$W(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit

Lock-Based Concurrency Control

Strict Two-Phase Locking (Strict 2PL)

Interleaved:

T1 obtains a shared lock on A
and reads A
T2 obtains a shared lock on A
and reads A
commits, locks are released
T2's lock request is now granted
T2 obtains an exclusive lock on B
and reads and writes B, commits
T1 obtains an exclusive lock on C
and reads and writes C, commits



$T1$	$T2$
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	

Lock-Based Concurrency Control

Strict Two-Phase Locking (Strict 2PL)

conflict equivalent: schedules that involve the actions of the same transactions and order every pair of conflicting actions of two committed transactions in the same way (operate on the same data object and at least one of them is a write)

conflict serializable: a schedule that is conflict equivalent to some serial schedule.

Lock-Based Concurrency Control

Strict Two-Phase Locking (Strict 2PL)

- some serializable schedules are not conflict serializable

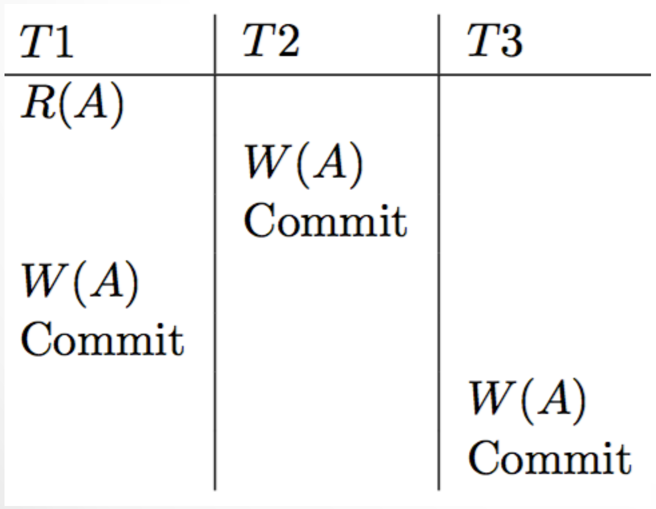
$T1$	$T2$	$T3$
$R(A)$		
	$W(A)$	
	Commit	
$W(A)$		
Commit		
		$W(A)$
		Commit

schedule is equivalent to executing the transactions serially in the order T1, T2, T3, but it is not conflict equivalent to this serial schedule because the writes of T1 and T2 are ordered differently

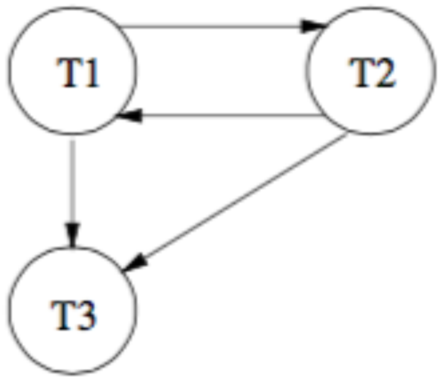
Lock-Based Concurrency Control

Precendence/serializability graph: captures all potential conflicts between the transactions in a schedule and contains:

- node for each committed transaction in S
- arc from T_i to T_j if an action of T_i precedes and conflicts with one of T_j's



$T1$	$T2$	$T3$
$R(A)$		
	$W(A)$	
	Commit	
$W(A)$		
Commit		
		$W(A)$
		Commit

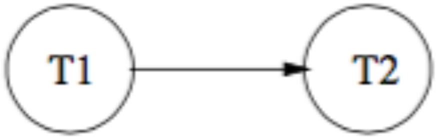


$$\begin{tikzcd}
T1 \arrow[r, shift left] \arrow[d] & T2 \arrow[l, shift left] \arrow[dl] \\
T3 &
\end{tikzcd}$$

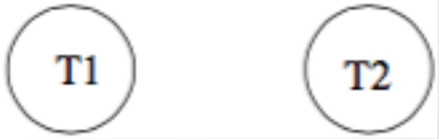
Lock-Based Concurrency Control

Precendence/serializability graphs

$T1$	$T2$
$X(A)$	
$R(A)$	
$W(A)$	
$X(B)$	
$R(B)$	
$W(B)$	
Commit	
	$X(A)$
	$R(A)$
	$W(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit

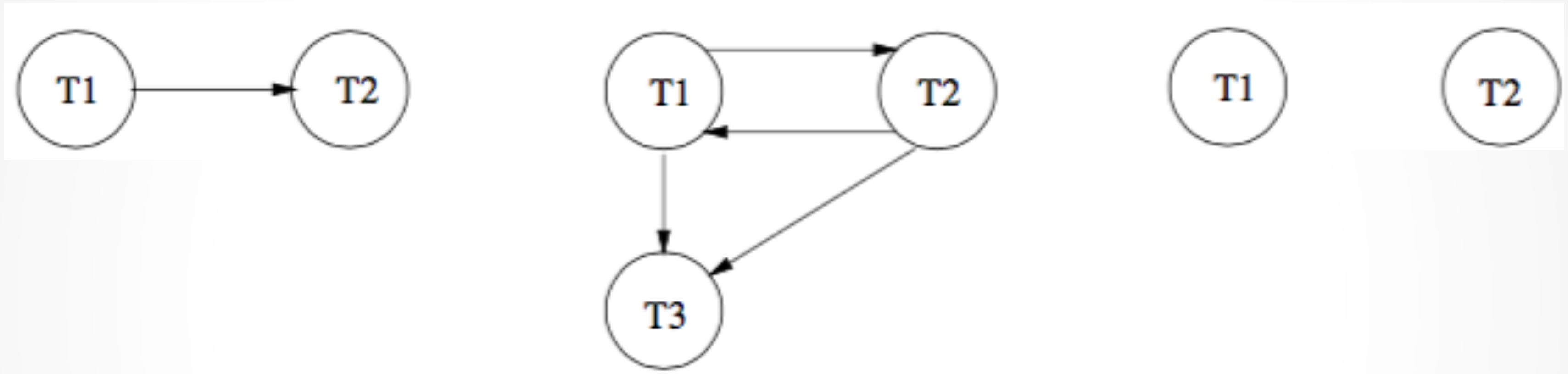


$T1$	$T2$
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	



Lock-Based Concurrency Control

Precendence/serializability graphs



The Strict 2PL protocol allows only serializable schedules:

- schedule S is conflict serializable if and only if its precedence graph is acyclic
- Strict 2PL ensures that the precedence graph for any schedule that it allows is acyclic

Lock-Based Concurrency Control

Two-Phase Locking (2PL) is a variant of Strict 2PL

relaxes the second rule of Strict 2PL - allows transactions to release locks before the end (before commit or abort action)

Rule(1) If T wants to read (and modify) an object, it first requests a shared (and then exclusive) lock on the object

Rule(2) A transaction cannot request additional locks once it releases *any* lock

every transaction has a 'growing' phase in which it acquires locks, followed by a 'shrinking' phase in which it releases locks

2PL ensures acyclicity of the precedence graph and therefore allows only serializable schedules

Lock-Based Concurrency Control

strict schedule: if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits.

Strict schedules are

- recoverable
- do not require cascading aborts
- actions of aborted transactions can be undone by restoring the original values of modified objects

Strict 2PL improves upon 2PL by guaranteeing that every allowed schedule is strict, in addition to being conflict serializable

Lock-Based Concurrency Control

View Serializability: two schedules S1 and S2 over the same set of transactions (any transaction that appears in either S1 or S2 must also appear in the other) are view equivalent under these conditions:

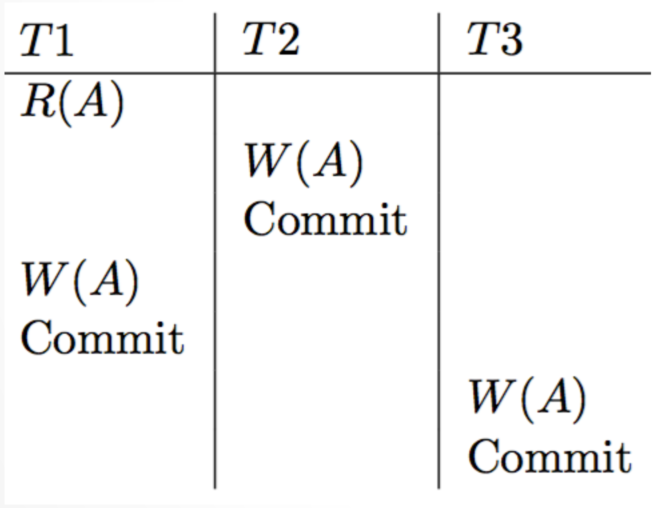
1. If T_i reads the initial value of object A in S1, it must also read the initial value of A in S2
2. If T_i reads a value of A written by T_j in S1, it must also read the value of A written by T_j in S2
3. For each data object A, the transaction (if any) that performs the final write on A in S1 must also perform the final write on A in S2

- schedule is view serializable if it is view equivalent to some serial schedule

Lock-Based Concurrency Control

View Serializability:

schedule is view serializable if it is view equivalent to some serial schedule



$T1$	$T2$	$T3$
$R(A)$		
	$W(A)$	
	Commit	
$W(A)$		
Commit		
		$W(A)$
		Commit

$$\begin{tikzcd}
T1 \arrow[r, bend left] \arrow[d] & T2 \arrow[l, bend left] \arrow[dl] \\
T3 &
\end{tikzcd}$$

- view serializable, although it is not conflict serializable
- any view serializable schedule that is not conflict serializable contains a blind write

Lock-Based Concurrency Control

Lock Management

lock manager: keeps track of the locks issued to transactions through a lock table

lock table: hash table with the data object identifier as the key

lock table entry: for an object(page/record) contains

- # of transactions currently holding a lock on the object (can be more than one if object is locked in shared mode)
- nature of the lock (shared or exclusive)
- a pointer to a queue of lock requests.

transaction table: descriptive entry for each transaction and the entry contains a pointer to a list of locks held by the transaction

Lock-Based Concurrency Control

Lock Requests

a transaction needs a lock on an object - issues a lock request to the lock manager:

- shared lock:
  - if the queue of requests is empty, and the object is not currently locked in exclusive mode -> grant the lock , update the lock table entry, increment # of transactions
- exclusive lock:
  - if no transaction currently holds a lock on the object (queue of requests is empty) -> grant the lock and update the lock table entry
- Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object
- transaction requesting the lock is suspended

Example: T1 has a shared lock on O, and T2 requests an exclusive lock, T2's request is queued. T3 requests a shared lock, its request enters the queue behind that of T2, (even though the requested lock is compatible with the lock held by T1)

Lock-Based Concurrency Control

Unlock Requests

All locks are released when a transaction aborts or commits

When a lock on an object is released:

- lock manager updates the lock table entry
- examines the lock request at the head of the queue

If this request can now be granted:

- transaction that made the request is woken up
- lock is granted.

If there are several requests for a shared lock on the object at the front of the queue, all of these requests can now be granted together

Lock-Based Concurrency Control

- **Lock Upgrades, Convoys, Latches**
- **lock upgrade:** transaction may need to acquire an exclusive lock on an object for which it already holds a shared lock
    - grant the write lock immediately if no other transaction holds a shared lock on the object
    - or insert the request at the front of the queue

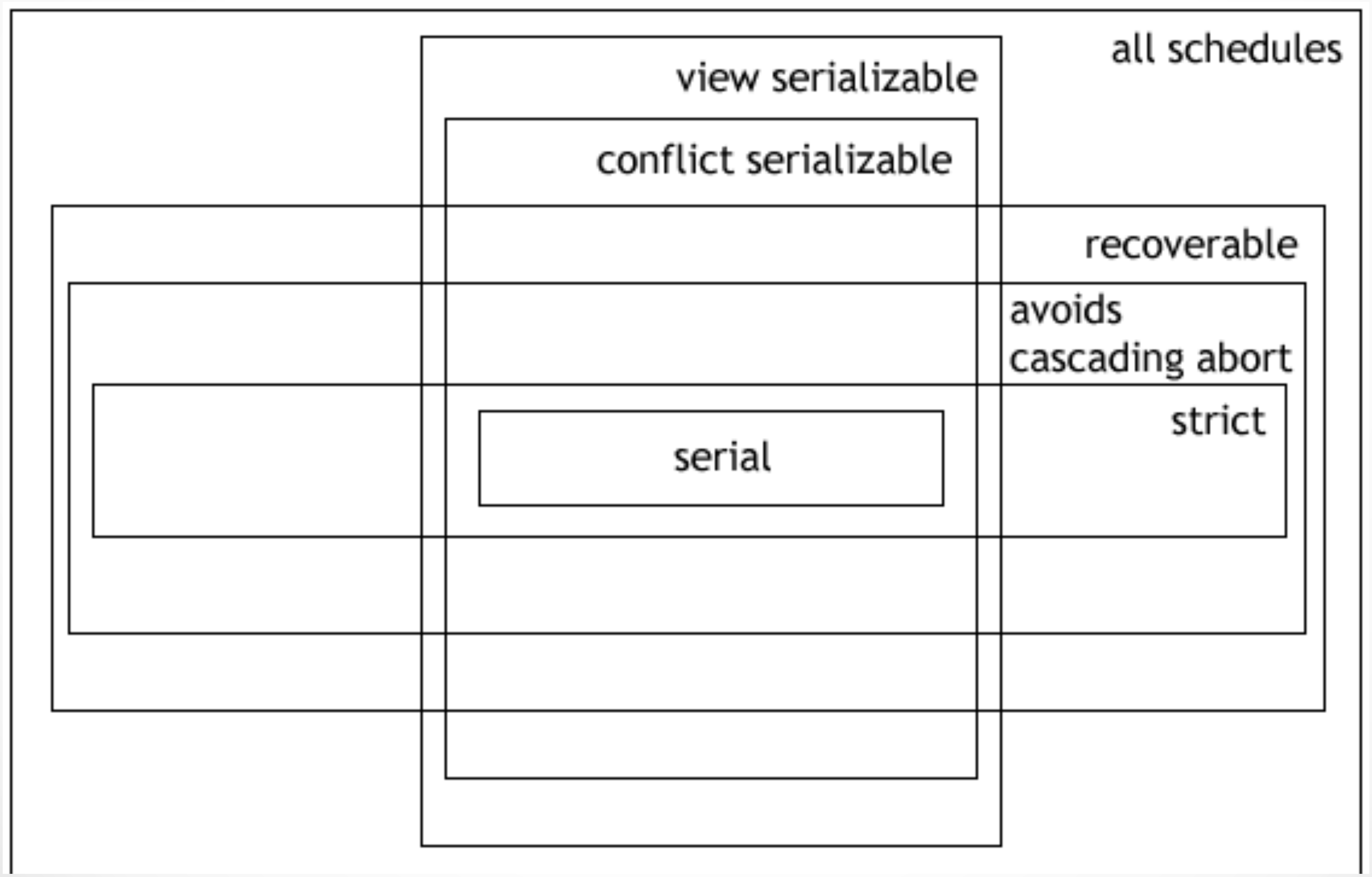
- **convoy**: interleaving interacting with the operating system's scheduling of processes' access to the CPU

Example: T holding a heavily used lock may be suspended by the operating system. Until T is resumed, every other transaction that needs this lock is queued (convoy)

- **latches**: short duration, setting a latch before reading or writing a page ensures that the physical read or write operation is atomic

Lock-Based Concurrency Control

Venn Diagram for Serializability and Recoverability Classes



Lock-Based Concurrency Control

- **Deadlocks**

Example:

T1 gets an exclusive lock on object A

T2 gets an exclusive lock on B

T1 requests an exclusive lock on B and is queued

T2 requests an exclusive lock on A and is queued

T1 is waiting for T2 to release its lock and T2 is waiting for T1 to release its lock

There may be other transactions waiting for these locks

Lock-Based Concurrency Control

Deadlock Prevention

- give priority to transactions using a startup **timestamp**

- lower the timestamp, the higher the transaction's priority, (oldest transaction has the highest priority)

T_i requests a lock and transaction T_j holds a conflicting lock:

- Wait-die: If T_i has higher priority, it is allowed to wait, otherwise it is aborted (lower priority transactions can never wait for higher priority transactions)

- Wound-wait: If T_i has higher priority, abort *Tj*, otherwise T_i waits (higher priority transactions never wait for lower priority transactions)

Lock-Based Concurrency Control

Deadlock Prevention

- give priority to transactions using a startup **timestamp**

When a transaction is aborted and restarted:

- given the same timestamp that it had originally (ensures that it will become the oldest transaction with the highest priority)
- get the locks that it requires

wait-die scheme is nonpreemptive - only a transaction requesting a lock can be aborted

wound-wait is preemptive

Lock-Based Concurrency Control

Deadlock Detection

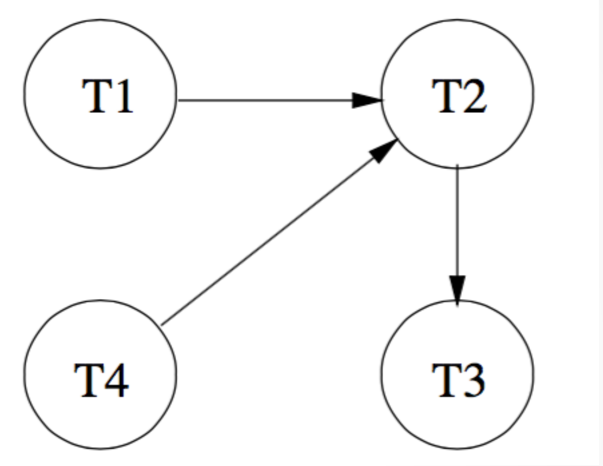
- DBMS must periodically check for deadlocks

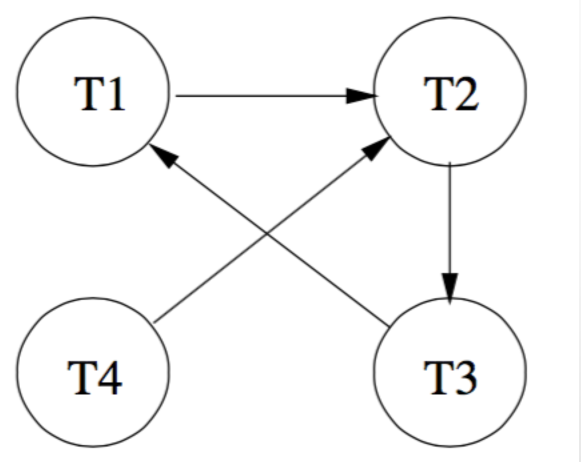
- **waits-for graph** –structures to detect deadlock cycles
  - nodes correspond to active transactions
  - there is an arc from Ti to Tj if (and only if) Ti is waiting for Tj to release a lock
  - edges are added when lock requests are queued
  - edges are removed when lock requests are granted

Lock-Based Concurrency Control

waits-for graphs

$T1$	$T2$	$T3$	$T4$
$S(A)$			
$R(A)$			
	$X(B)$		
	$W(B)$		
$S(B)$			
		$S(C)$	
		$R(C)$	
	$X(C)$		
			$X(B)$
		$X(A)$	





X(A) creates a cycle in the waits-for graph (deadlock)

Lock-Based Concurrency Control

Deadlock Detection

- waits-for graphs are periodically checked for cycles (deadlocks)

- alternative to waits-for graphs is using timeout mechanism (if transaction is waiting too long for a lock, assume deadlock and abort)

Lock-Based Concurrency Performance

- Detection vs Prevention?

Lock-Based Concurrency Control

Lock-Based Concurrency Performance

- conflicts resolved either by:
  - Blocking: force transactions to wait
  - Aborting: wastes work done by that transaction

- concurrency control by:
  - Prevention: preemptive abort mechanism, good for heavy lock contention
  - Detection: deadlock cycle of transactions hold locks, system throughput reduced, better since deadlocks are relatively infrequent

- **Conservative 2PL**: variant of 2PL prevents deadlocks – transaction obtains all locks at the beginning (or blocks waiting)
  - Tradeoff: locks are acquired earlier (good for heavy lock contention because average lock time is reduced, transactions are never blocked)

Lock-Based Concurrency Control

Lock-Based Concurrency Performance

Deadlock transaction victim can be chosen based on:

- fewest locks
- least work
- farthest from completion
- least restarted transaction

designing a good concurrency control mechanism is complex

Lock-Based Concurrency Control

Dynamic Databases and the Phantom Problem

T1: scans Sailors relation for oldest sailor for each rating level 1 & 2 by:

- locking all pages with rating 1 sailors
- finds age of oldest sailor (71)

T2: inserts new sailor with rating 1 and age 96

- this record is inserted in another page not containing sailors with rating 1 so no exclusive lock conflict with T1

locks page with oldest sailor (80) with rating 2 and deletes this sailor, commits, releases locks

T1: locks all remaining pages with sailors with rating 2, gets age of oldest sailor - 63

- Both T1 and T2 follow Strict 2PL and commit

Lock-Based Concurrency Control

Dynamic Databases and the Phantom Problem

Both T1 and T2 follow Strict 2PL and commit

- flaw is not in the Strict 2PL protocol
    - locking pages that contain records at a given time does not prevent new "phantom" records from being added on other pages
- Strict 2PL guarantees conflict serializability (no cycles in the precedence graph)
- but objects that should have been locked by T1 was altered by the actions of T2, outcome differs that of any serial execution
- if new items are added to the database, conflict serializability does not guarantee serializability

Lock-Based Concurrency Control

Dynamic Databases and the Phantom Problem

Solution:

1. if there is no index and all pages in the file must be scanned
   - T1 must ensure that no new pages are added to the file
   - lock all existing pages

2. Index locking: if there is a dense index on the *rating* field
   - T1 can obtain a lock on the index page that contains a data entry with rating=1
   - If there are no such data entries
     - the page that would contain a data entry for rating=1 is locked
     - transaction that tries to insert a record with rating=1 into the Sailors relation is blocked until T1 releases its locks

predicate locking: more general concept than index locking, supports implicit locking of all records that match an arbitrary predicate (expensive to implement)

Lock-Based Concurrency Control

Concurrency Control in B+ Trees

- we could ignore the index structure and treat each page as a data object, and use some version of 2PL
  - would lead to very high lock contention in the higher levels of the tree as every tree search begins at the root

Observe:

1. higher levels of the tree only serve to direct searches, and all the 'real' data is in the leaf levels, searches should be shared locks, lock can be released once child node lock is obtained
2. for inserts, a node must be locked in exclusive mode only if a split can propagate up to it from the modified leaf

Lock-Based Concurrency Control

Concurrency Control in B+ Trees

2. for inserts, a node must be locked in exclusive mode only if a split can propagate up to it from the modified leaf

- conservative: exclusive locks on all nodes from the root to the leaf node to be modified (splits can propagate all the way from a leaf to the root)
- or lock a child node, release the lock on the parent if the child is not full

Lock-Based Concurrency Control

Concurrency Control in B+ Trees

search for data entry 38*

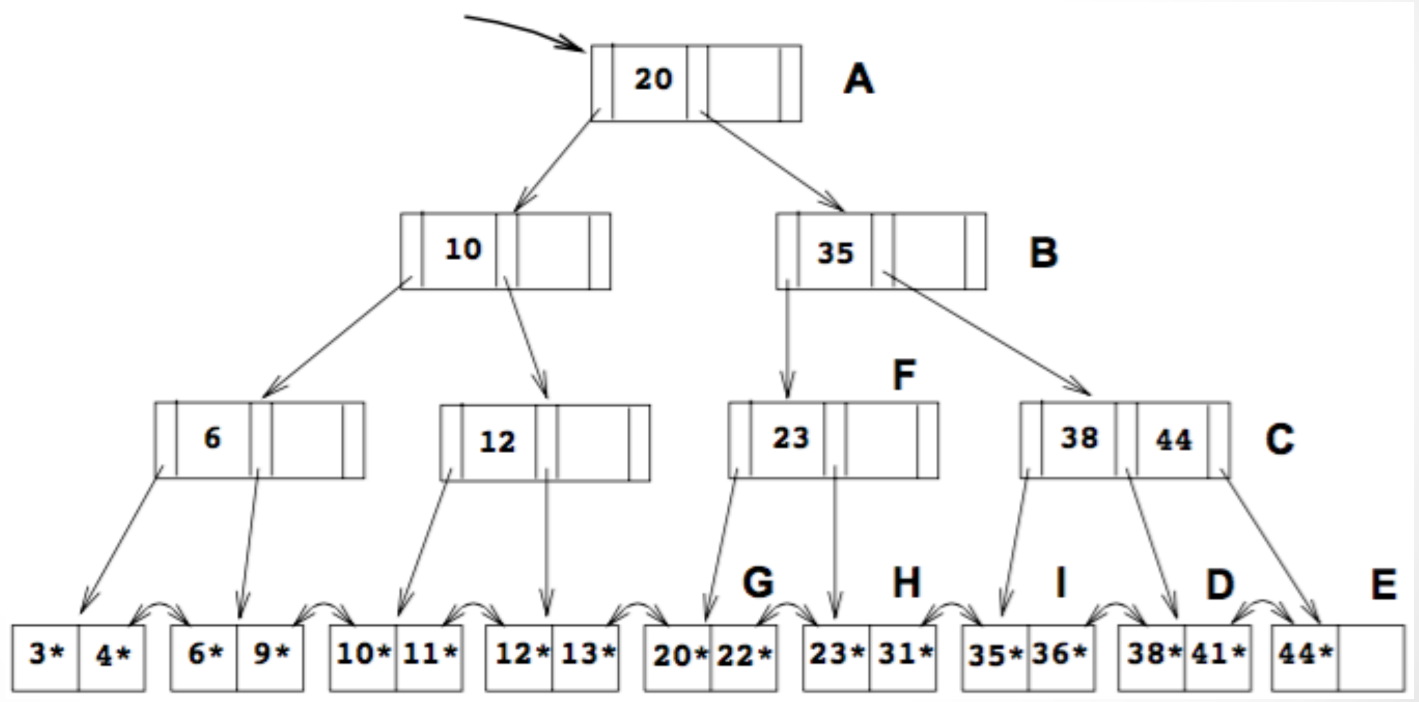
T_i

~~S(A)~~

~~S(B)~~

~~S(C)~~

S(D)



Lock-Based Concurrency Control

Concurrency Control in B+ Trees

insert data entry 45*

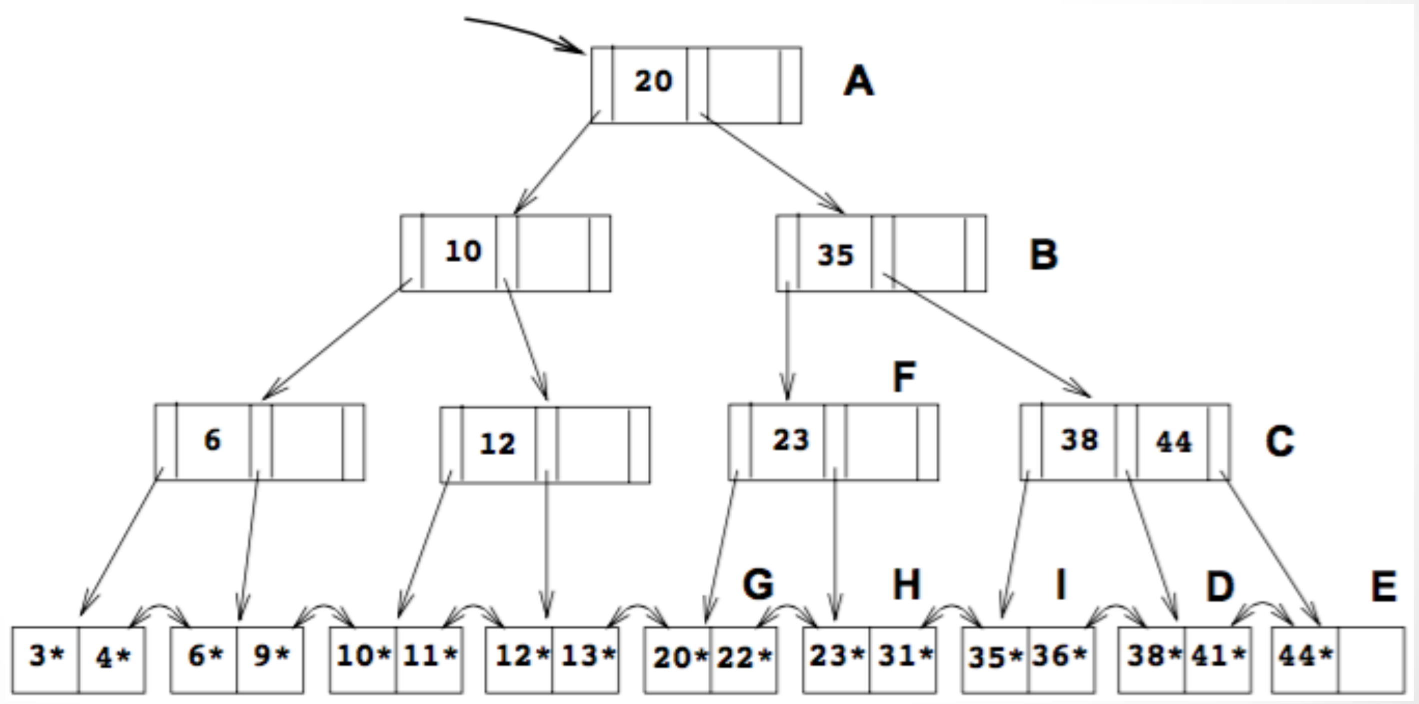
T_i

~~S(A)~~

~~S(B)~~

~~S(C)~~

X(E)

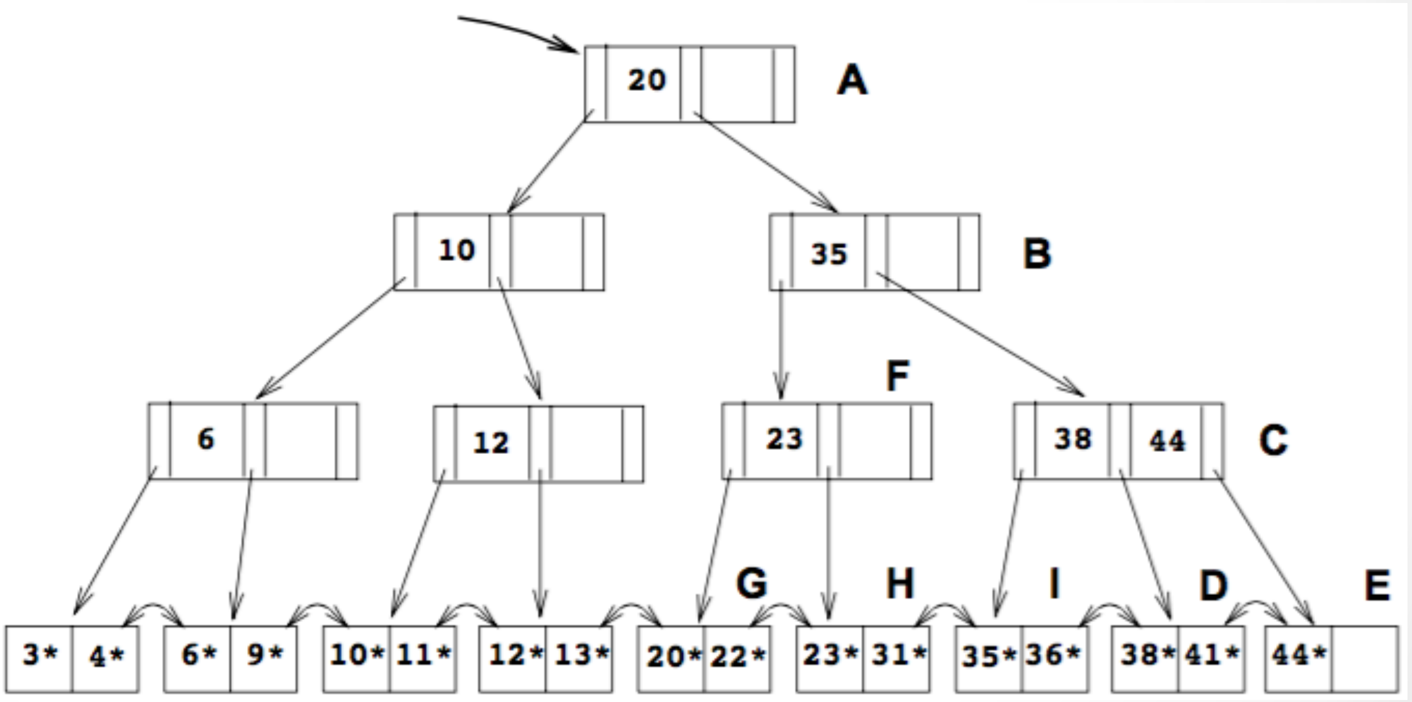


Lock-Based Concurrency Control

Concurrency Control in B+ Trees

insert data entry 25*

T_i
S(A)
~~S(B)~~
~~S(F)~~
X(H)



- to improve performance, obtain shared locks except for the leaf which is locked in exclusive mode
- if leaf is full, however, we must upgrade from shared locks to exclusive locks for all nodes to which the split propagates

Lock-Based Concurrency Control

Transaction Support in SQL-92

- A transaction is automatically started when a user executes a statement that modifies either the database or the catalogs:
  - `SELECT` query
  - `UPDATE` command
  - `CREATE TABLE` statement

- Transaction is terminated by:
  - `COMMIT` command
  - ROLLBACK (the SQL keyword for abort) command

Lock-Based Concurrency Control

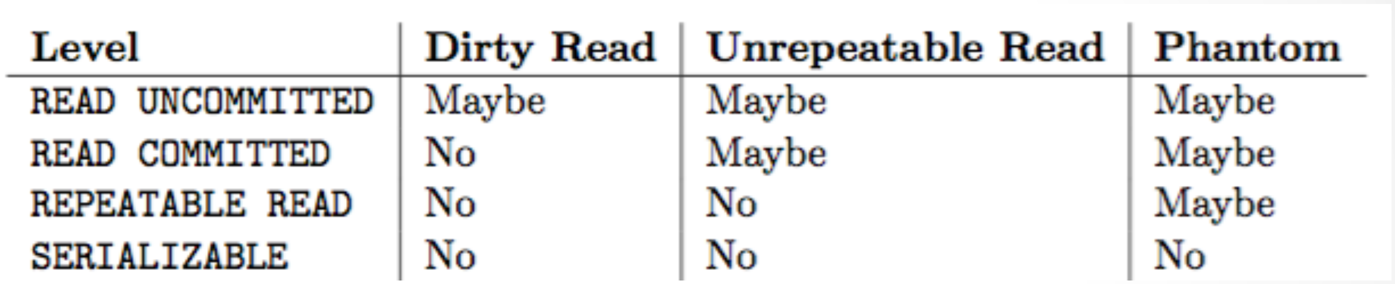
Transaction Characteristics

1. **access mode**:
   - `READ ONLY` - transaction is not allowed to modify the database, only shared locks need to be obtained
   - `READ WRITE – INSERT, DELETE, UPDATE`, and `CREATE` commands can be executed
2. **diagnostic size**: determines the number of error conditions that can be recorded
3. **isolation level**: controls to which extent a given transaction is exposed to the actions of other transactions executing concurrently
   - `READ UNCOMMITTED`
   - `READ COMMITTED`
   - `REPEATABLE READ`
   - `SERIALIZABLE`

Lock-Based Concurrency Control

Transaction Characteristics

isolation level: controls to which extent a given transaction is exposed to the actions of other transactions executing concurrently



Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

`SERIALIZABLE` – highest degree of isolation, T reads only the changes made by committed transactions, obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged

`REPEATABLE READ` – T reads only the changes made by committed transactions, T could experience the phantom phenomenon, same locking protocol as above, does not do index locking (only individual objects)

`READ COMMITTED` – same as above, a value read by T may well be modified by another transaction while T is still in progress, exclusive locks before writing, shared before reading (released immediately)

`READ UNCOMMITTED` – transaction T can read changes made to an object by an ongoing transaction, the object can be changed further while T is in progress, no shared or exclusive locks, has access mode of `READ ONLY`

Lock-Based Concurrency Control

Transaction Characteristics

isolation level and access mode can be set using the `SET TRANSACTION` command

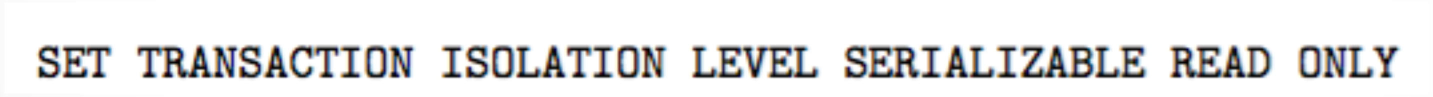
```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
```

when a transaction is started, the default is `SERIALIZABLE` and `READ WRITE`

Lock-Based Concurrency Control

Transaction Characteristics

isolation level and access mode can be set using the `SET TRANSACTION` command

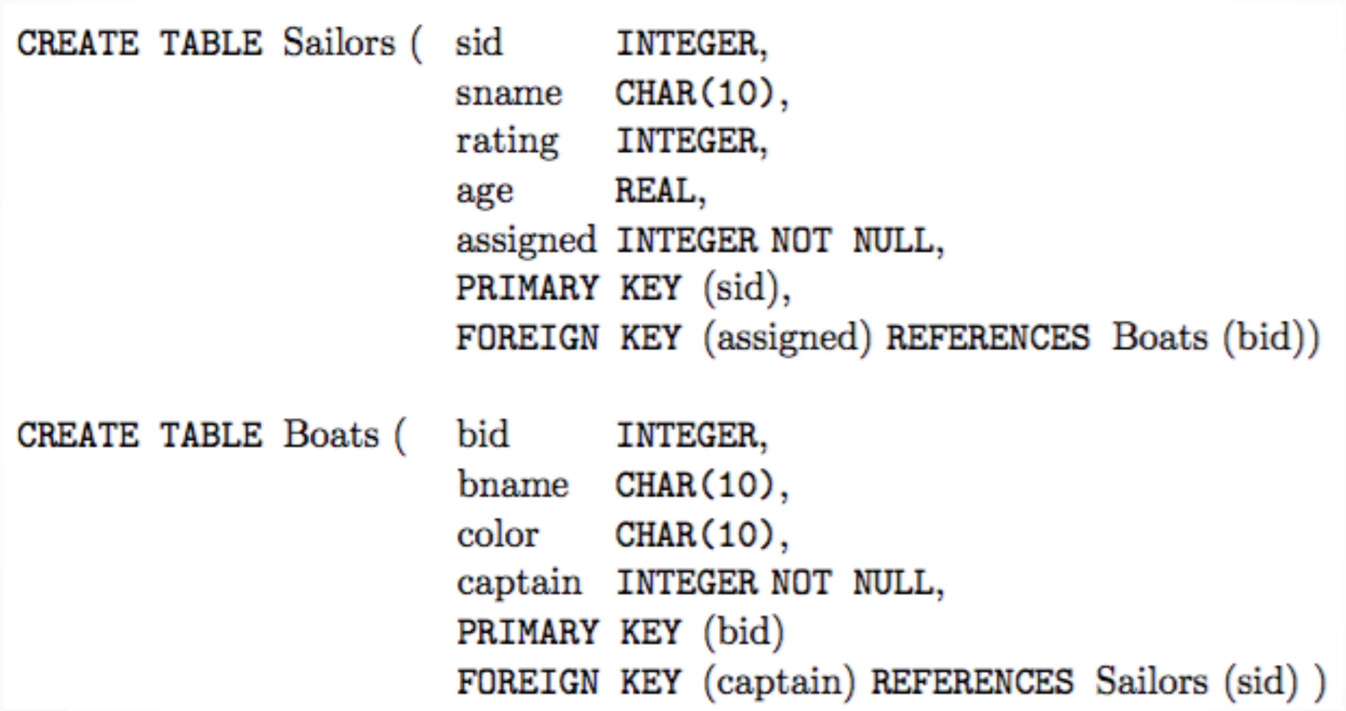


```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
```

when a transaction is started, the default is `SERIALIZABLE` and `READ WRITE`

Lock-Based Concurrency Control

Transactions and Constraints



```
CREATE TABLE Sailors ( sid      INTEGER,
                       sname    CHAR(10),
                       rating   INTEGER,
                       age      REAL,
                       assigned INTEGER NOT NULL,
                       PRIMARY KEY (sid),
                       FOREIGN KEY (assigned) REFERENCES Boats (bid))

CREATE TABLE Boats (   bid      INTEGER,
                       bname    CHAR(10),
                       color    CHAR(10),
                       captain  INTEGER NOT NULL,
                       PRIMARY KEY (bid)
                       FOREIGN KEY (captain) REFERENCES Sailors (sid) )
```

How to insert the very first boat or sailor tuple?

```
SET CONTRAINT <name> DEFERRED
```

SUMMARY

- **conflict equivalent** - if two schedules order every pair of conflicting actions of two committed transactions in the same way
- **conflict serializable** schedule is conflict equivalent to some serial schedule
- **strict schedule** - if a value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits
- **precedence graph or serializability graph** describe potential conflicts between transactions in a schedule
- two-phase locking (2PL) allows transactions to release locks before the transaction commits or aborts
- **2PL** and Strict **2PL** ensure that only conflict serializable schedules are permitted to execute
- **lock manager** maintains a **lock table** with **lock table entries**
- **deadlock** is a cycle of transactions that are all waiting for another transaction in the cycle to release a lock.
- **conservative 2PL** is a deadlock-preventing locking scheme
- **phantom problem** can be avoided through index locking