

Transactions, Recovery and Concurrency (II)

Concurrency Control

Concurrency Control Methods

- **Locking Mechanism**

The idea of locking some data item X is to:

- give a transaction exclusive use of the data item X ,
- do not restrict the access of other data items.

This prevents one transaction from changing a data item currently being used in another transaction.

- We will discuss a simple locking scheme which locks individual items, using read and write locks

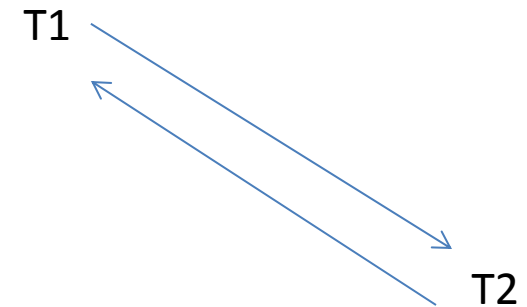
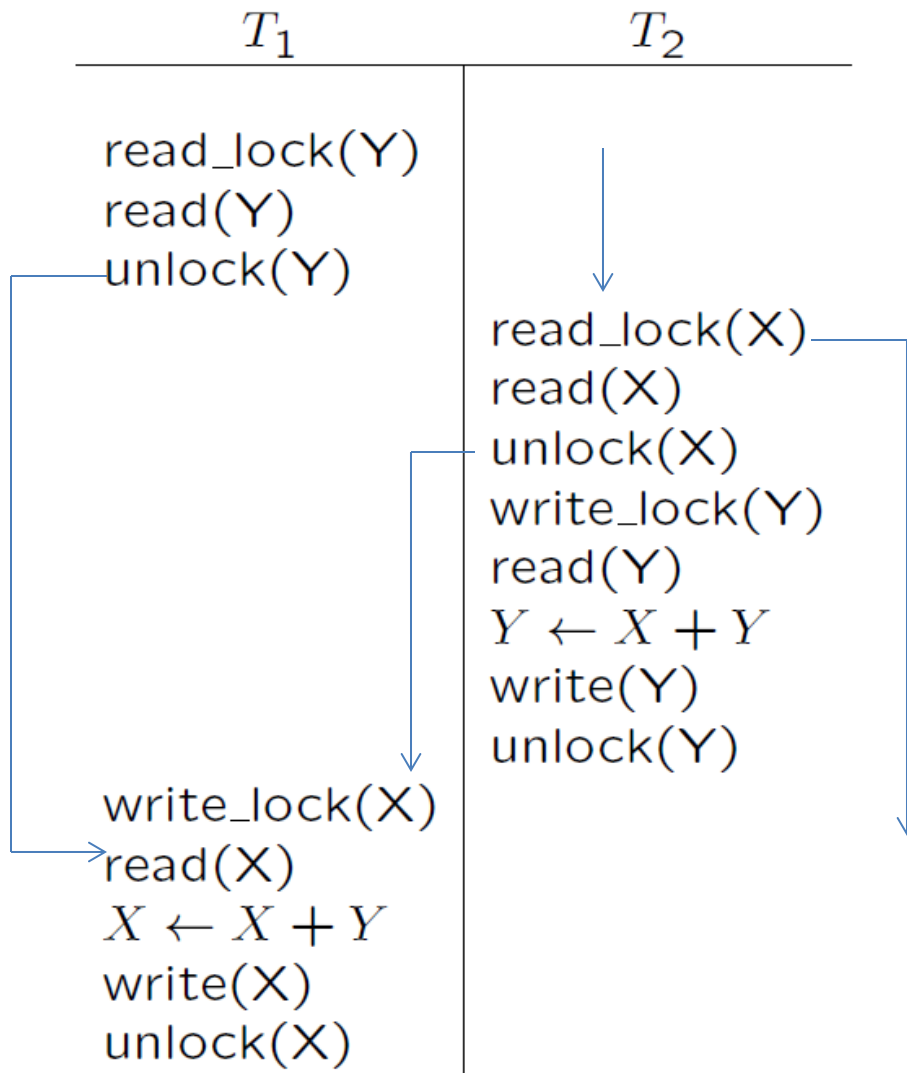
Locking Rules

- In this schema, every transaction T must obey the following rules.
- 1) If T has only one operation (read/write) manipulating an item X :
 - obtain a read lock on X before reading it,
 - obtain a write lock on X before writing it,
 - unlock X when done with it.
- 2) If T has several operations manipulating X :
 - obtain one proper lock only on X :
a read lock if all operations on X are reads;
a write lock if one of these operations on X is a write.
 - unlock X after the last operation on X in T has been executed.

Locking Rules (cont.)

- In this scheme,
 - Several read locks can be issued on the same data item at the same time.
 - A read lock and a write lock cannot be issued on the same data item at the same time, neither two write locks.
- This still does not guarantee serializability.

Example: Based on E/N Fig 18.3.



Two Phase Locking (2PL)

- To guarantee serializability, transactions must also obey the *two-phase locking protocol*:
 - Growing Phase: all locks for a transaction must be obtained before any locks are released, and
 - Shrinking Phase: gradually release all locks (once a lock is released no new locks may be requested).

Two Phase Locking (2PL) (Cont.)

Example: Based on E/N Fig 18.4.

T_1

read_lock(Y)
read(Y)
write_lock(X)
unlock(Y)
read(X)
$X \leftarrow X + Y$
write(X)
unlock(X)

- Locking thus provides a solution to the problem of correctness of schedules.

Two phase locking ensures conflict serializability

Deadlock

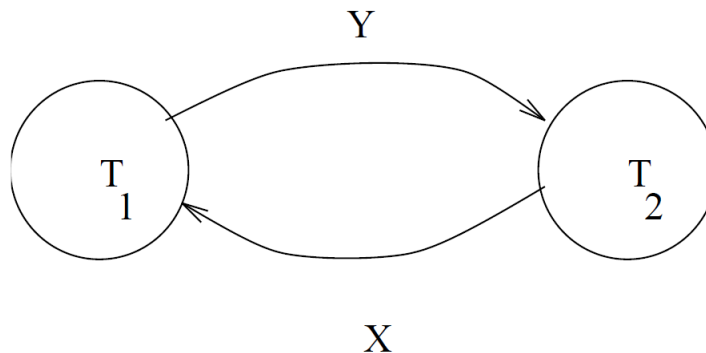
- A problem that arises with locking is **deadlock**.
- Deadlock occurs when two transactions are each waiting for a lock on an item held by the other.

T_1	T_2
write_lock(X) read(X)	
write_lock(Y) **waiting for Y*** **waiting for Y***	write_lock(Y) read(Y)
	write_lock(X) ***waiting for X***

Deadlock Check

- Create the *wait-for graph* for currently active transactions:
 - create a vertex for each transaction; and
 - an arc from T_i to T_j if T_i is waiting for an item locked by T_j .
- If the graph has a cycle, then a *deadlock* has occurred.

Example:



Several methods to deal with deadlocks

- deadlock detection
 - periodically check for deadlocks, abort and rollback some transactions (restart them later). This is a good choice if transactions are very short or very independent.

Several methods to deal with deadlocks (Cont.)

- deadlock prevention - Assign priorities based on timestamps.
Assume T_i wants a lock that T_j holds. Two policies are possible:
 - Wait-Die: If T_i has higher priority, T_i waits for T_j ; otherwise T_i aborts
 - Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits
- If a transaction re-starts, make sure it has its original timestamp

Timestamp ordering

- The idea here is:
 - to assign each transaction a timestamp (e.g. start time of transaction), and
 - to ensure that the schedule used is equivalent to executing the transactions in timestamp order

- Each data item, X , is assigned
 - a read timestamp, $read\ TS(X)$ - the latest timestamp of a transaction that read X , and
 - a write timestamp, $write\ TS(X)$ - the latest timestamp of a transaction that write X .

- These are used in read and write operations as follows. Suppose the transaction timestamp is T .

read(X):

```
If T >= write_TS(X) then
    { execute read(X);
      if T >= read_TS(X) then
        read_TS(X) <- T }
else
    rollback the transaction and restart
```

write(X):

```
If T >= read_TS(X) and T >= write_TS(X) then
    { execute write(X); write_TS(X) <- T }
else
    rollback and restart
```

- **Thomas' write rule:**

write(X):

```
If T < read_TS(X) then
    rollback and restart
else if T < write_TS(X) then
    ignore the write
else
    { execute write(X);
      write_TS(X) <- T }
```

T_1
read (x)

T_2
read (y)
write (y)

read (z)

T_3

read (z)
write (z)

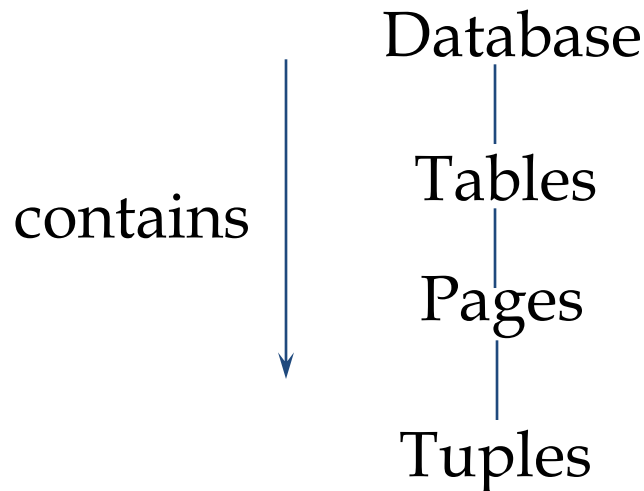
Write (z)

$r_TS(x) = 0 \rightarrow 1$
 $w_TS(x) = 0$
 $r_TS(y) = 0 \rightarrow 2$
 $w_TS(y) = 0 \rightarrow 2$
 $r_TS(z) = 0 \rightarrow 1 \rightarrow 3$
 $w_TS(z) = 0 \rightarrow 3$

- Some problems:
 - Cyclic restart: There is no deadlock, but a kind of livelock can occur - some transactions may be constantly aborted and restarted.
 - Cascading rollback: When a transaction is rolled back, so are any transactions which read a value written by it, and any transactions which read a value written by them . . . etc. This can be avoided by not allowing transactions to read values written by uncommitted transactions (make them wait).

Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to decide!
- Data “containers” are nested:



Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new “**intention**” locks:
 - ❖ Before locking an item, Xact must set “intention locks” on all its ancestors.
 - ❖ For unlock, go from specific to general (i.e., bottom-up).
 - ❖ **SIX mode**: Like S & IX at the same time.

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Multiple Granularity Lock Protocol

- Each Xact starts from the root of the hierarchy.
- To get S or X lock on a node, must hold IS or IX on parent node.
 - What if Xact holds SIX on parent? S on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Examples

- T1 scans R, and updates a few tuples:
 - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
 - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
 - T3 gets an S lock on R.
 - OR, T3 could behave like T2; can use **lock escalation** to decide which.

	--	IS	IX	S	X
--	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	
IX	✓	✓	✓		
S	✓	✓		✓	
X	✓				

Dynamic Databases

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)

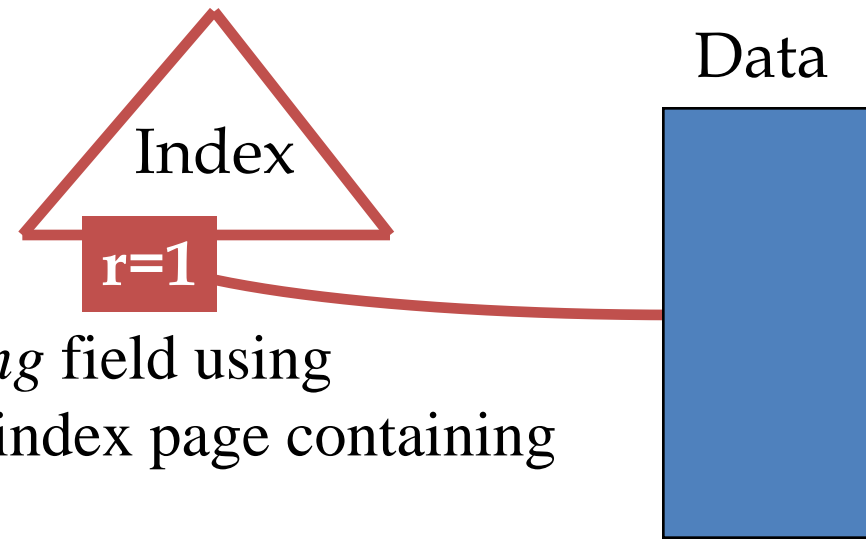
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
 - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
 - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
 - T2 also deletes oldest sailor with *rating* = 2 (and, say, *age* = 80), and commits.
 - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- No consistent DB state; however T1 “correctly” gets through!

The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
 - Assumption only holds if no sailor records are added while T1 is executing!
 - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

Index Locking



- If there is a dense index on the *rating* field using Alternative (2), T1 should lock the index page containing the data entries with *rating* = 1.
 - If there are no records with *rating* = 1, T1 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T1 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no new records with *rating* = 1 are added.

Predicate Locking

- Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
- What is the predicate in the sailor example?
- In general, predicate locking has a lot of locking overhead.

Locking in B+ Trees

- How can we efficiently lock a particular leaf node?
 - Btw, don't confuse this with multiple granularity locking!
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
 - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.

Two Useful Observations

- Higher levels of the tree only direct searches for leaf pages.
- For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)
- We can exploit these observations to design efficient locking protocols that guarantee serializability *even though they violate 2PL.*

A Simple Tree Locking Algorithm

- **Search:** Start at root and go down; repeatedly, S lock child then unlock parent.
- **Insert/Delete:** Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:
 - If child is safe, release all locks on ancestors.
- **Safe node:** Node such that changes will not propagate up beyond this node.
 - Inserts: Node is not full.
 - Deletes: Node is not half-empty.

ROOT



A

Do:

- 1) Search 38*
- 2) Delete 38*
- 3) Insert 45*
- 4) Insert 25*



B



F



C

G

H

I

D

E



A Better Tree Locking Algorithm (See Bayer-Schkolnick paper)

- **Search:** As before.
- **Insert/Delete:**
 - Set locks as if for search, get to leaf, and set X lock on leaf.
 - If leaf is not **safe**, release all locks, and restart Xact using previous Insert/Delete protocol.
- Gambles that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful. In practice, better than previous alg.

ROOT

F

20

A

35

B

23

F

38

44

C

G

H

I

D

E

20*

22*

23*

24*

35*

36*

38*

41*

44*

Do:

1) Delete 38*

2) Insert 25*

4) Insert 45*

5) Insert 45*,
then 46*

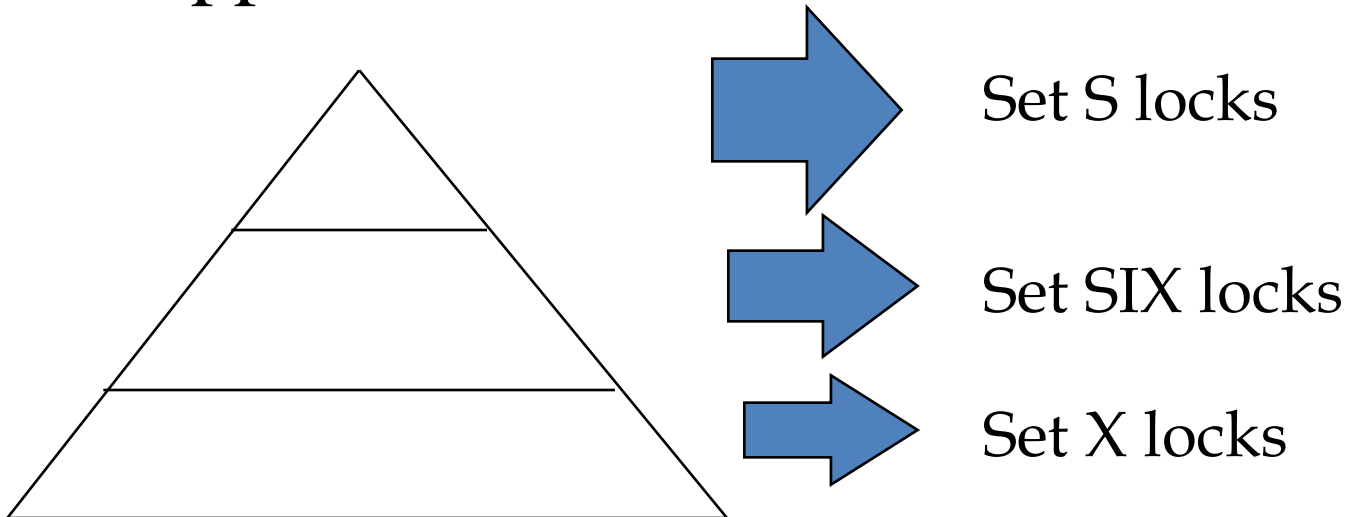
Even Better Algorithm

- **Search:** As before.
- **Insert/Delete:**
 - Use original Insert/Delete protocol, but set IX locks instead of X locks at all nodes.
 - Once leaf is locked, convert all IX locks to X locks **top-down**: i.e., starting from node nearest to root. (Top-down reduces chances of deadlock.)

(Contrast use of IX locks here with their use in multiple-granularity locking.)

Hybrid Algorithm

- The likelihood that we really need an X lock decreases as we move up the tree.
- Hybrid approach:



Multiversioning

- Similar to the timestamp ordering approach; but is allowed to access “old” versions of a table.
- A history of the values and timestamps (versions) of each item is kept.
- When the value of an item is needed, the system chooses a **proper** version of the item that maintains serializability.
- This results in fewer aborted transactions at the cost of greater complexity to maintain more versions of each item.

- We will look at a scheme, several versions X_1, \dots, X_k of each data item are kept. For each X_i we also keep
 - *read* $TS(X_i)$ - as for timestamp ordering.
 - *write* $TS(X_i)$ - as for timestamp ordering.

- Read and write are done as follows for a transaction P with timestamp T .

read(X):

```
Find  $X_i$  s.t.  $\text{write\_TS}(X_i)$  is the  
    highest write timestamp but  $\leq T$   
update  $\text{read\_TS}(X_i)$  (and do  $\text{read}(X_i)$ )  
return  $X_i$  as the value for  $X$ 
```

write(X):

```
Find Xi s.t. write_TS(Xi) is the
    highest write timestamp but <= T
if T < read_TS(Xi) then
    rollback and restart
else
    { create a new version X(k+1) of X;
      set read_TS(X(k+1)) to T;
      set write_TS(X(k+1)) to T}
```

- *Note:* Cascading rollback and cyclic restart problems can still occur, but should be reduced.
- However, there is an increased overhead in maintaining multiple versions of items.

Optimistic scheduling

- In two-phase locking, timestamp ordering, and multiversioning concurrency control techniques, a certain degree of checking is done **before** a database operation can be executed.
- The idea here is to push on and hope for the best!
- No checking is done while the transaction is executing.

- The protocol has three phases.
 - read phase - A transaction can read data items from the database into local variables. However, updates are applied only to local copies of the data items kept in the transaction workspace.
 - validation phase - checks are made to ensure that serializability is not violated,
 - write phase -if validation succeeds, updates are applied and the transaction is committed. Otherwise, the updates are discarded and the transaction is restarted.

- A scheme uses timestamps and keeps each transaction's
 - read-set - the set of items read by the transaction,
 - write-set - the set of items written by the transaction.
- During validation, we check that the transaction does not interfere with any transaction that is committed or currently validating.

- Each transaction T is assigned 3 timestamps:
 $Start(T)$, $Validation(T)$, $Finish(T)$.
- To pass the validation test for T , one of the following must be true:
 - 1. $Finish(S) < Start(T)$; or
 - 2. for S s.t. $Start(T) < Finish(S)$, then
 - a) write set of S is disjoint from the read set of T , and
 - b) $Finish(S) < Validation(T)$.

- Optimistic control is a good option if there is not much interaction between transactions.
- **Note:** Our earlier treatment of recovery methods largely ignored concurrency issues.

2PL vs. TSO vs. MV vs. OP

- A Comparison among two-phase locking (2PL), timestamp ordering (TSO), multiversioning (MV), optimistic (OP) concurrency control techniques.
- MV should provide the greatest concurrency degree (in average). However, we need to maintain multiversions for each data item.
- 2PL can offer the second greatest concurrency degree (in average); but will result in deadlocks. To resolve the deadlocks, either
 - need additional computation to detect deadlocks and to resolve the deadlocks, or
 - reduce the concurrency degree to prevent deadlocks by adding other restrictions.

2PL vs. TSO vs. MV vs. OP (cont.)

- If most transactions are very short, we can use 2PL + deadlock detection and resolution.
- TSO has a less concurrency degree than that of 2PL if a proper deadlock resolution is found. However, TSO does not cause deadlocks. Other problems, such as cyclic restart and cascading rollback, will appear in TSO.
- If there are not much interaction between transactions, OP is a very good choice. Otherwise, OP is a bad choice.