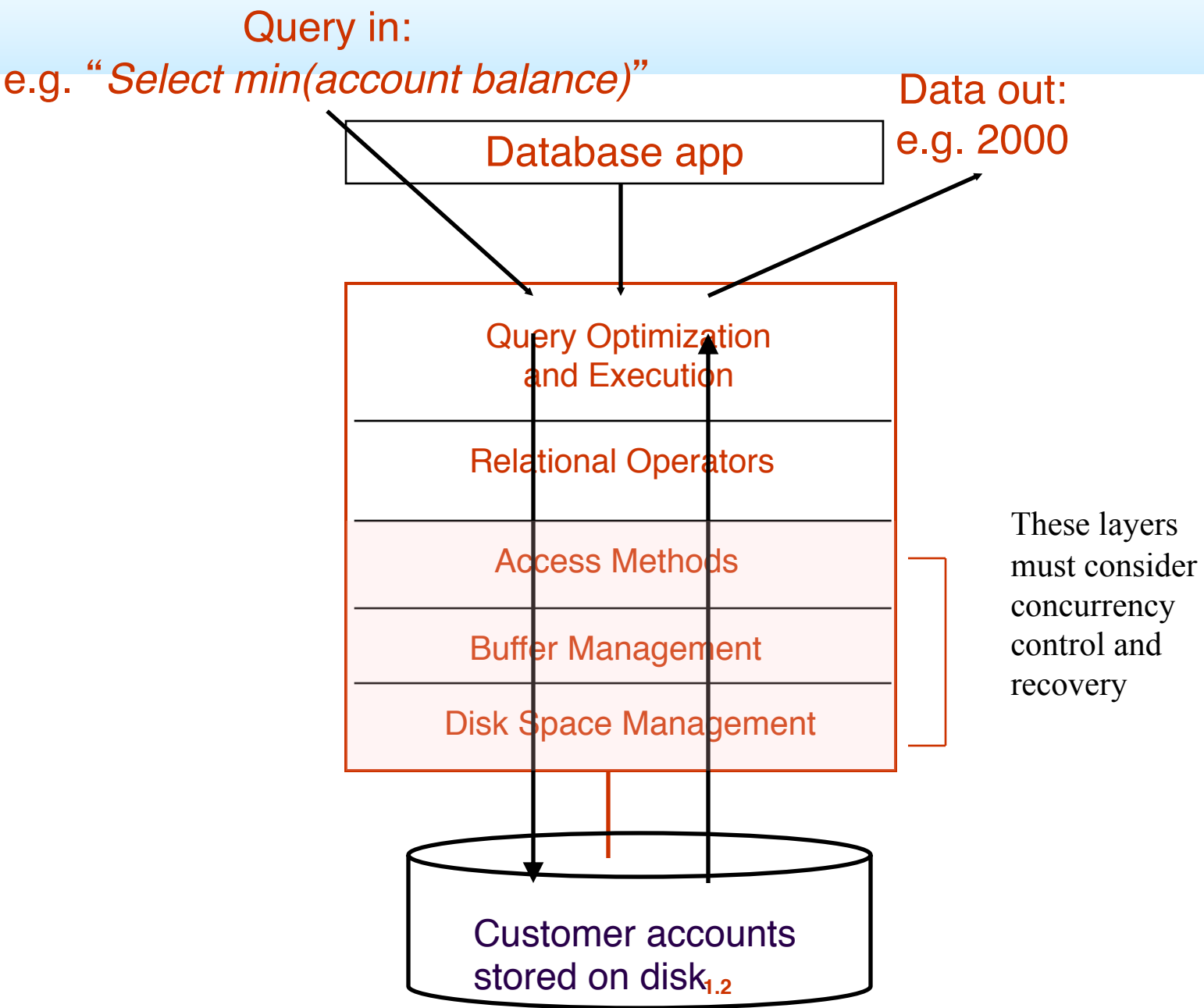
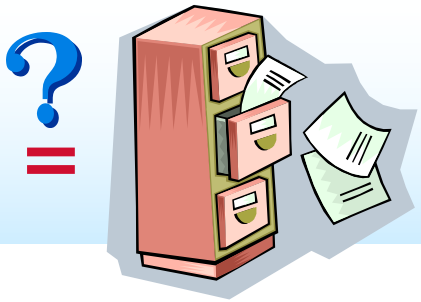
CAS CS 660
Introduction to Database Systems

Transactions
and
Concurrency Control

Recall: Structure of a DBMS





File System vs. DBMS?

- Thought Experiment 1:
  - You and your project partner are editing the same file.
  - You both save it at the same time.
  - Whose changes survive?

A) Yours B) Partner's C) Both D) Neither E) ???

- **Thought Experiment 2:**
  - –You're updating a file.
  - –The power goes out.
  - –Which of your changes survive?

Q: How do you write programs over a subsystem when it promises you only "???" ?

A: Very, very carefully!!

A) All B) None C) All Since last save D) ???

Concurrent Execution

- Concurrent execution essential for good performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user queries concurrently.
  - Trends are towards lots of cores and lots of disks.
    - e.g., IBM Watson has 2880 processing cores
- A program may carry out many operations, but the DBMS is only concerned about what data is read/written from/to the database.

Key concept: Transaction

- an atomic sequence of database actions (reads/writes)
- takes DB from one consistent state to another
- *transaction* - DBMS's abstract view of a user program:
  - a sequence of reads and writes.

Example

- Here, *consistency* is based on our knowledge of banking "semantics"
- In general, up to writer of transaction to ensure transaction preserves consistency
- DBMS provides (limited) automatic enforcement, via integrity constraints
  - e.g., balances must be >= 0

Transaction - Example

```
BEGIN;    --BEGIN TRANSACTION
  UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';

  UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts
    WHERE name = 'Alice');

  UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';

  UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts
    WHERE name = 'Bob');

COMMIT;    --COMMIT WORK
```

Transaction Example (with Savepoint)

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';

SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';

-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;

UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

The ACID properties of Transactions

There are three side effects of acid. Enhanced long term memory, decreased short term memory, and I forget the third.
- Timothy Leary

- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of all others.
- **D** urability: If a transaction commits, its effects persist.

Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.

- Atomic Transactions: a user can think of a transaction as always either executing all its actions, or not executing any actions at all.
  - One approach: DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
  - Another approach: *Shadow Pages*
  - Logs won because of need for audit trail and for efficiency reasons.

Transaction Consistency

- "Consistency" - data in DBMS is accurate in modeling real world, follows integrity constraints
- User must ensure transaction consistent by itself
- If DBMS is consistent before transaction, it will be after also.

- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).
  - DBMS enforces some ICs, depending on the ICs declared in CREATE TABLE statements.
  - Beyond this, DBMS does not understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

Isolation (Concurrency)

- Multiple users can submit transactions.
- Each transaction executes as if it was running by itself.
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- We will formalize this notion shortly.
- Many techniques have been developed. Fall into two basic categories:
  - Pessimistic – don't let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them *after* they happen.

Durability - Recovering From a Crash

- System Crash - short-term memory (RAM) lost (disk okay)
  - This is the case we will handle.
- Disk Crash - "stable" data lost
  - ouch --- need back ups; raid-techniques can help avoid this.
- **There are 3 phases in Aries recovery (and most others):**
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, as found in the log), working backwards in the log.
- **At the end --- all committed updates and only those updates are reflected in the database.**
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

Plan of attack (ACID properties)

- First we'll deal with "I", by focusing on concurrency control.
- Then we'll address "A" and "D" by looking at recovery.
- What about "C"?
  - Well, if you have the other three working, and you set up your integrity constraints correctly, then you get this for free (!?).

Example

- **Consider two transactions (*Xacts*):**

```
T1:     BEGIN   A=A+100,   B=B-100   END
T2:     BEGIN   A=1.06*A,   B=1.06*B   END
```

- 1st xact transfers $100 from B's account to A's
- 2nd credits both accounts with 6% interest.
- Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2???
  - $2000 *1.06 = $2120
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

- **Legal outcomes: A=1166,B=954 or A=1160,B=960**
- **Consider a possible interleaved *schedule*:**

T1:	A=A+100		B=B-100
T2:		A=1.06*A	B=1.06*B

- This is OK (same as T1;T2). But what about:

T1:	A=A+100		B=B-100
T2:		A=1.06*A, B=1.06*B	

- **Result: A=1166, B=960; A+B = 2126, bank loses $6**
- **The DBMS's view of the second schedule:**

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

Scheduling Transactions

- *Serial schedule:* A schedule that does not interleave the actions of different transactions.
  - i.e., you run the transactions serially (one at a time)

- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) and output of executing the first schedule is identical to the effect of executing the second schedule.

- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time.

Anomalies with Interleaved Execution

Unrepeatable Reads:

```
T1:    R(A),                          R(A), W(A), C
T2:            R(A), W(A), C
```

Reading Uncommitted Data (“dirty reads”):

```
T1:    R(A), W(A)                     R(B), W(B), Abort
T2:            R(A), W(A), C
```

Overwriting Uncommitted Data:

```
T1:    W(A)                     W(B), C
T2:            W(A), W(B), C
```

Conflict Serializable Schedules

- We need a formal notion of equivalence that can be implemented efficiently…
- Two operations conflict if they are by different transactions, they are on the same object, and at least one of them is a write.
- Two schedules are conflict equivalent iff:

  They involve the same actions of the same transactions, and

  every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.
- Note, some "serializable" schedules are NOT conflict serializable.
  - This is the price we pay for efficiency.

Dependency Graph

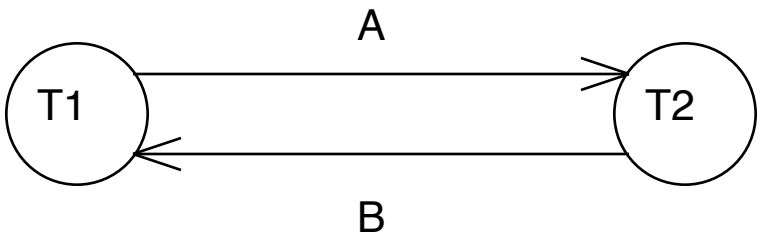
- *Dependency graph*: One node per Xact; edge from *Ti* to *Tj* if an operation of Ti conflicts with an operation of Tj and Ti's operation appears earlier in the schedule than the conflicting operation of Tj.

- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

Example

- A schedule that is not conflict serializable:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	



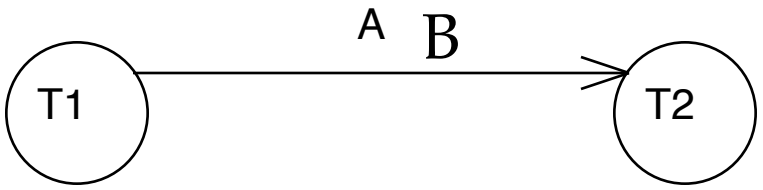
Dependency graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Example

- A schedule that is conflict serializable:

T1:	R(A), W(A),		R(B), W(B),
T2:		R(A), W(A),	R(B), W(B)

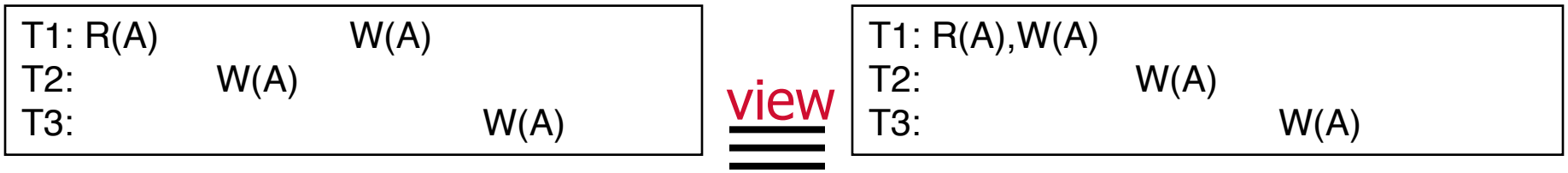


Dependency graph

- No Cycle Here!

View Serializability – an Aside

- Alternative (weaker) notion of serializability.
- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2
- Basically, allows all conflict serializable schedules + "blind writes"



Notes on Conflict Serializability

- Conflict Serializability doesn't allow all schedules that you would consider correct.
  - This is because it is strictly *syntactic* - it doesn't consider the meanings of the operations or the data.
- In practice, Conflict Serializability is what gets used, because it can be done efficiently.
  - Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, etc.
- Two-phase locking (2PL) is how we implement it.

Locks

- We use “locks” to control access to items.
- Shared (S) locks – multiple transactions can hold these on a particular item at the same time.
- Exclusive (X) locks – only one of these and no other locks, can be held on a particular item at a time.

Lock Compatibility Matrix

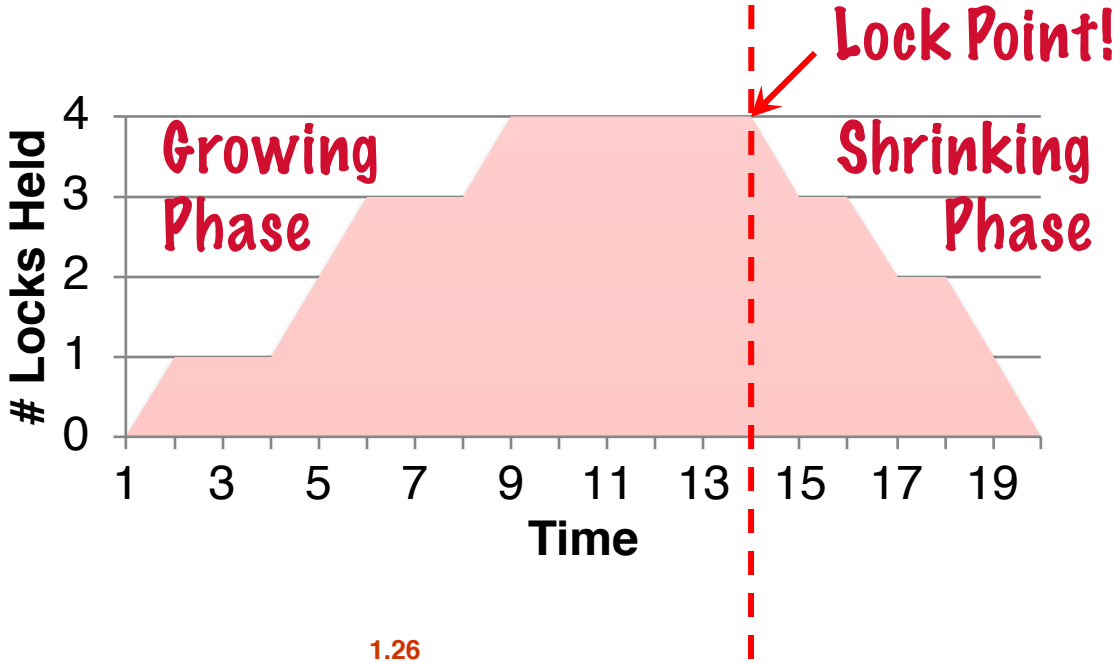
	S	X
S	√	–
X	–	–

Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - a S (*shared*) or an X (exclusive) lock on object before reading,
   - an X (*exclusive*) lock on object before writing.

2) A transaction can not request additional locks once it releases any locks.

Thus, each transaction has a "growing phase" followed by a "shrinking phase".



Two-Phase Locking (2PL)

2PL on its own is sufficient to guarantee conflict serializability.

- Doesn't allow dependency cycles! (note: see "Deadlock" discussion a few slides hence)
- Schedule of conflicting transactions is conflict equivalent to a serial schedule ordered by "lock point".

Ex 1: A= 1000, B=2000, Output =?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	<granted>
	Read(A)
	Unlock(A)
	Lock_S(B) <granted>
Lock_X(B)	
	Read(B)
<granted>	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

Is it a 2PL schedule?

No, and it is not serializable.

Ex 2: A= 1000, B=2000, Output =?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	<granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Is it a 2PL schedule? Yes: so it is serializable.

Avoiding Cascading Aborts – Strict 2PL

- **Problem with 2PL: Cascading Aborts**
- **Example: rollback of T1 requires rollback of T2!**

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A)	

- Solution: Strict Two-phase Locking (Strict 2PL):
  - Same as 2PL, except:
  - All locks held by a transaction are released only when the transaction completes

Strict 2PL (continued)

All locks held by a transaction are released only when the transaction completes

- Like 2PL, Strict 2PL allows only schedules whose precedence graph is acyclic, but it is actually stronger than needed for that purpose.

- In effect, "shrinking phase" is delayed until:
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (then locks can be released after rollback).

Ex 3: A= 1000, B=2000, Output =?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	<granted>
	Read(A)
	Lock_S(B) <granted>
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

Is it a 2PL schedule? Strict 2PL?

Ex 2: Revisited

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Unlock(A)	<granted>
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	<granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Is it Strict 2PL? No: Cascading Abort Poss.

Lock Management

- Lock and unlock requests are handled by the Lock Manager.
- LM contains an entry for each currently held lock.
- Lock table entry:
  - Ptr. to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to **queue of lock requests**
- When lock request arrives see if anyone else holds a conflicting lock.
  - If not, create an entry and grant the lock.
  - Else, put the requestor on the wait queue
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - Can cause deadlock problems

Ex 4: Output = ?

Lock_X(A) <granted>	
	Lock_S(B) <granted>
	Read(B)
	Lock_S(A)
Read(A)	
A: = A-50	
Write(A)	
Lock_X(B)	

Is it a 2PL schedule?

Strict 2PL?

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- **Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:**
  - Wait-Die: If Ti is older, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti is older, Tj aborts; otherwise Ti waits
- **If a transaction re-starts, make sure it gets its original timestamp**
  - Why?

Deadlock Detection

- Alternative is to allow deadlocks to happen but to check for them and fix them if found.

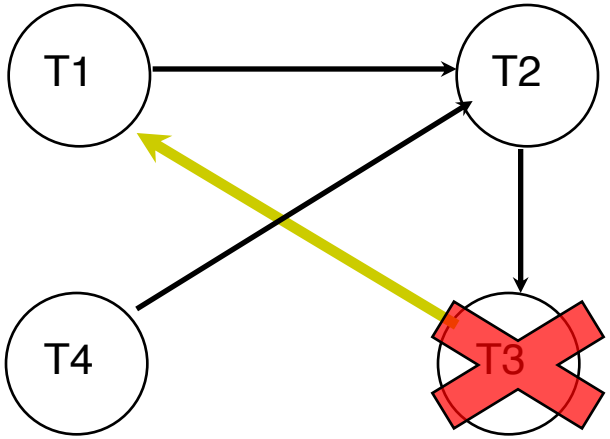
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

- If cycle detected – find a transaction whose removal will break the cycle and kill it.

Deadlock Detection (Continued)

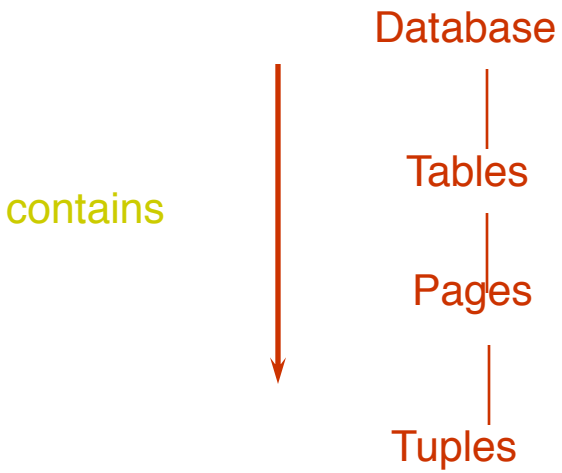
- Example:

- T1: S(A), S(D), S(B)
- T2: X(B) X(C)
- T3: S(D), S(C), X(A)
- T4: X(B)



Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to make same decision for all transactions!
- Data "containers" are nested:



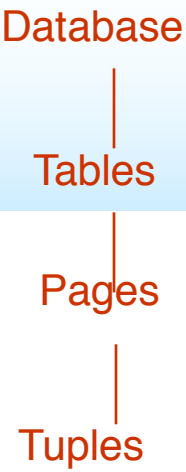
Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new "intention" locks:
- Still need S and X locks, but before locking an item, Xact must have proper intension locks on all its ancestors in the granularity hierarchy.

  - IS – Intent to get S lock(s) at finer granularity.
  - IX – Intent to get X lock(s) at finer granularity.
  - SIX mode: Like S & IX at the same time. Why useful?

	IS	IX	SIX	S	X
IS	√	√	√	√	-
IX	√	√	-	-	-
SIX	√	-	-	-	-
S	√	-	-	√	-
X	-	-	-	-	-

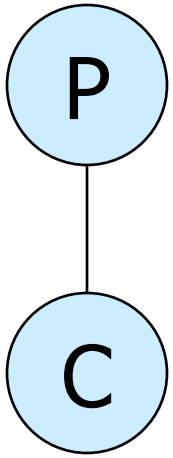
Multiple Granularity Lock Protocol



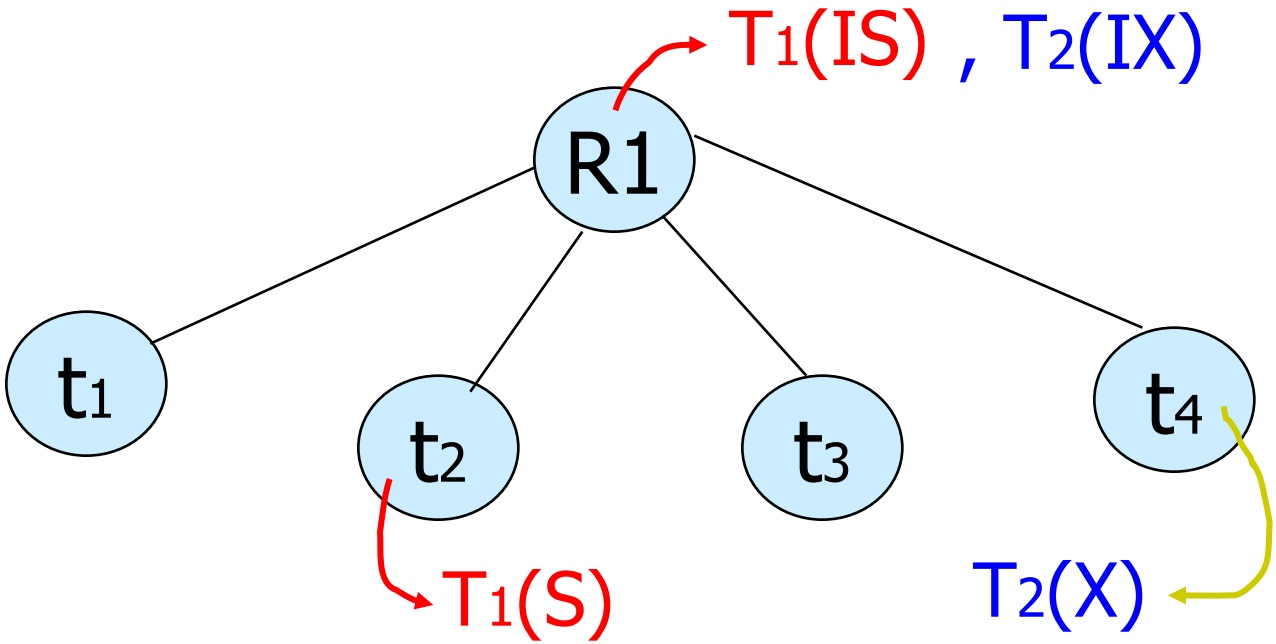
- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
  - What if Xact holds SIX on parent? S on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

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

Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X, SIX
S	[S, IS] not necessary
SIX	X, IX, [SIX]
X	none



Example



Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:

(1) Follow multiple granularity comp function

(2) Lock root of tree first, any mode

(3) Node Q can be locked by Ti in S or IS only if parent(Q) can be locked by Ti in IX or IS

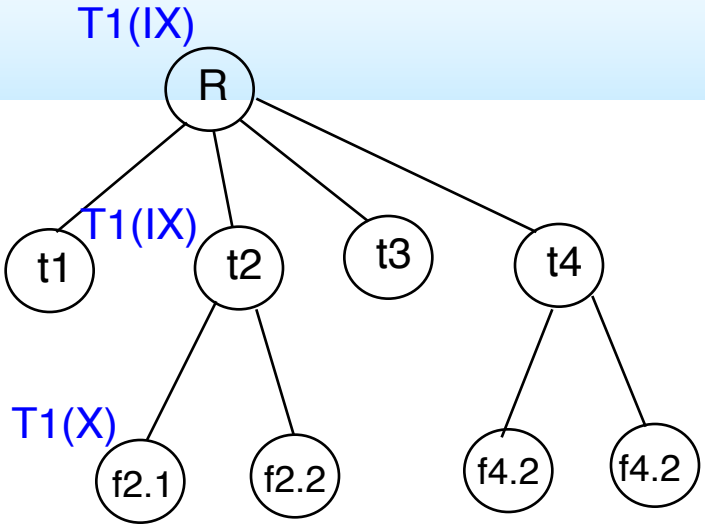
(4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX

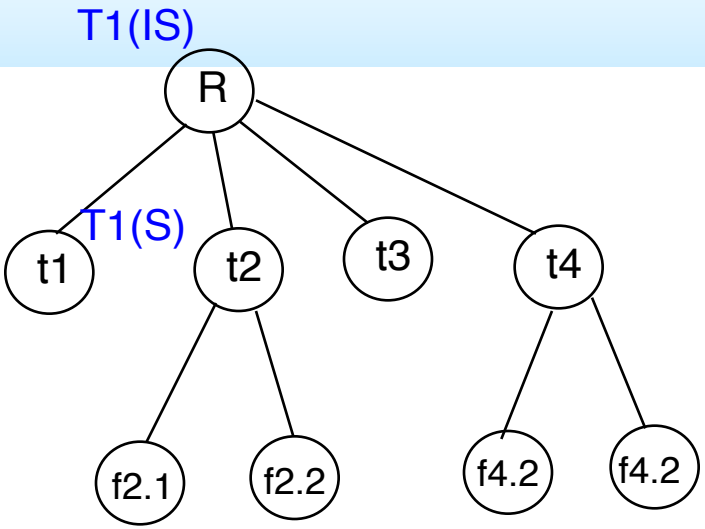
(5) Ti uses 2PL

(6) Ti can unlock node Q only if none of Q's children are locked by Ti

- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

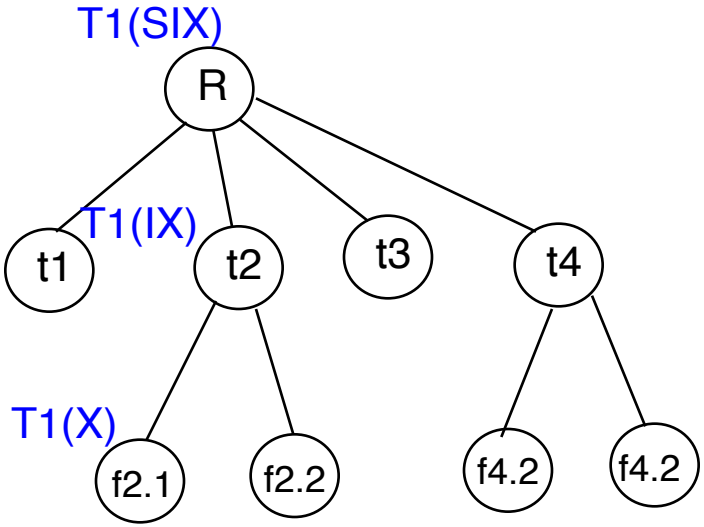
Examples





Can T2 access object f2.2 in X mode?

What locks will T2 get?



Examples – 2 level hierarchy

Tables

Tuples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- 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.
  - Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

The "Phantom" Problem

- With Insert and Delete, even Strict 2PL (on individual items) will not assure serializability:
- Consider T1 – "Find oldest sailor"
  - T1 locks all records, and finds oldest sailor (*age* = 71).
  - Next, T2 inserts a new sailor; *age* = 96 and commits.
  - T1 (within the same transaction) checks for the oldest sailor again and finds sailor aged 96!!
- The sailor with age 96 is a "phantom tuple" from T1's point of view --- first it's not there then it is.
- No serial execution where T1's result could happen!

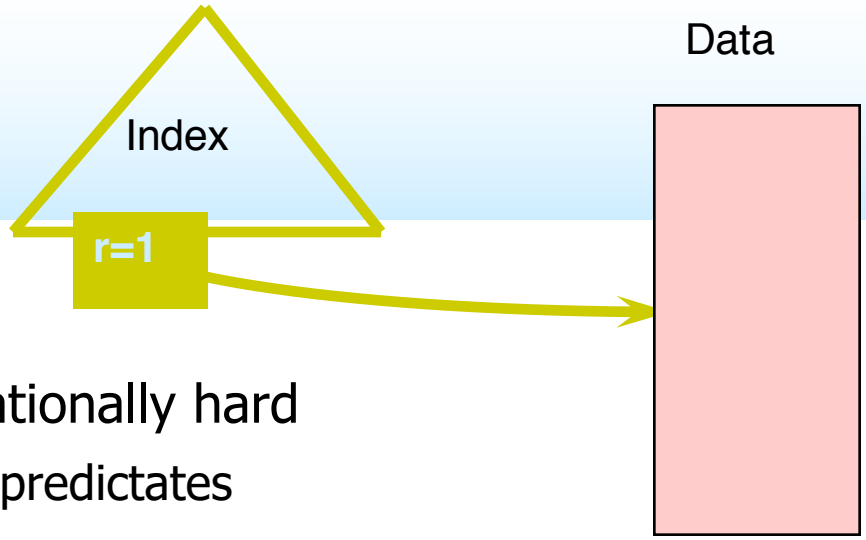
The "Phantom" Problem – example 2

- Consider T3 – "Find oldest sailor for each rating"
  - T3 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T4 inserts a new sailor; *rating* = 1, *age* = 96.
  - T4 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T3 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- T3 saw only part of T4's effects!
- No serial execution where T3's result could happen!

The Problem

- T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate.
  - Assumption only holds if no sailor records are added while they are executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Examples show that conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed!

Solution: Index Key Value Locking



- Locking Predicates directly is computationally hard
  - Need to calculate overlap of arbitrary predictates
- If there is an index on the *rating* field using Alternative (2), T3 should lock the index page containing the data entries with *rating* = 1.
  - If there are no records with *rating* = 1, T3 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T3 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no records with *rating* = 1 are added or deleted.

Isolation Levels

- SQL standard offers several isolation levels
  - Each transaction can have level set separately
  - Problematic definitions, but in best practice done with variations in lock holding
- Serializable
  - (ought to be default, but not so in practice)
  - Traditionally done with Commit-duration locks on data and indices (to avoid phantoms)
- Repeatable Read
  - Commit-duration locks on data
  - Phantoms can happen
- Read Committed
  - short duration read locks, commit-duration write locks
  - non-repeatable reads possible
- Read Uncommitted
  - no read locks, commit-duration write locks

Optimistic CC (Kung-Robinson)

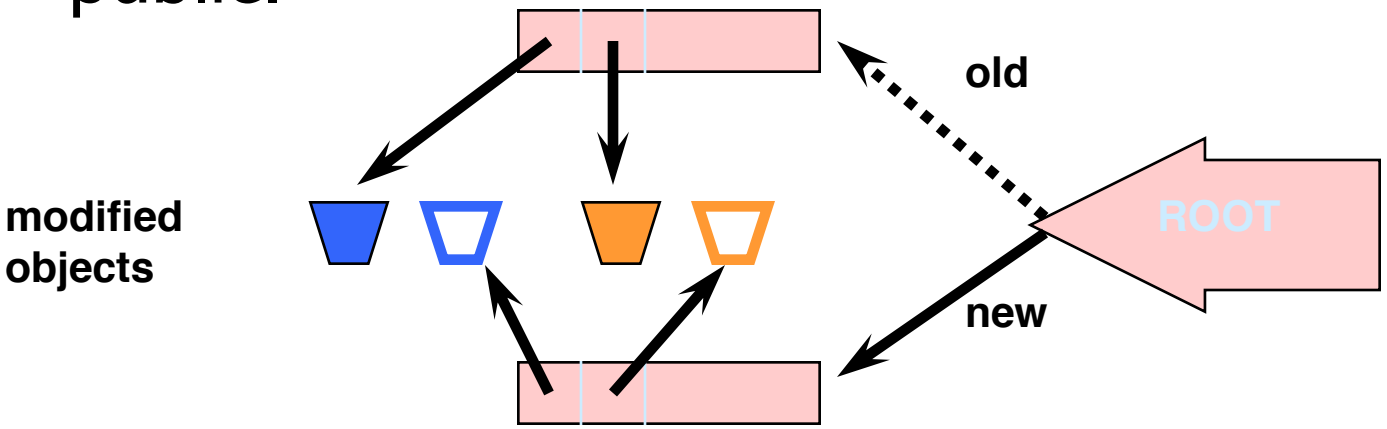
Locking is a conservative approach in which conflicts are prevented. Disadvantages:

- Lock management overhead.
- Deadlock detection/resolution.
- Lock contention for heavily used objects.

- Locking is "pessimistic" because it assumes that conflicts will happen.
- If conflicts are rare, we might get better performance by not locking, and instead checking for conflicts at commit.

Kung-Robinson Model

- Xacts have three phases:
  - READ: Xacts read from the database, but make changes to private copies of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.



Validation

- Test conditions that are sufficient to ensure that no conflict occurred.
- Each Xact is assigned a numeric id.
  - Just use a **timestamp (call it T_i)**.
- Timestamps are assigned at end of READ phase, just before validation begins.
- ReadSet(Ti): Set of objects read by Xact T_i
- WriteSet(Ti): Set of objects modified by T_i

Test 1 – non-overlapping

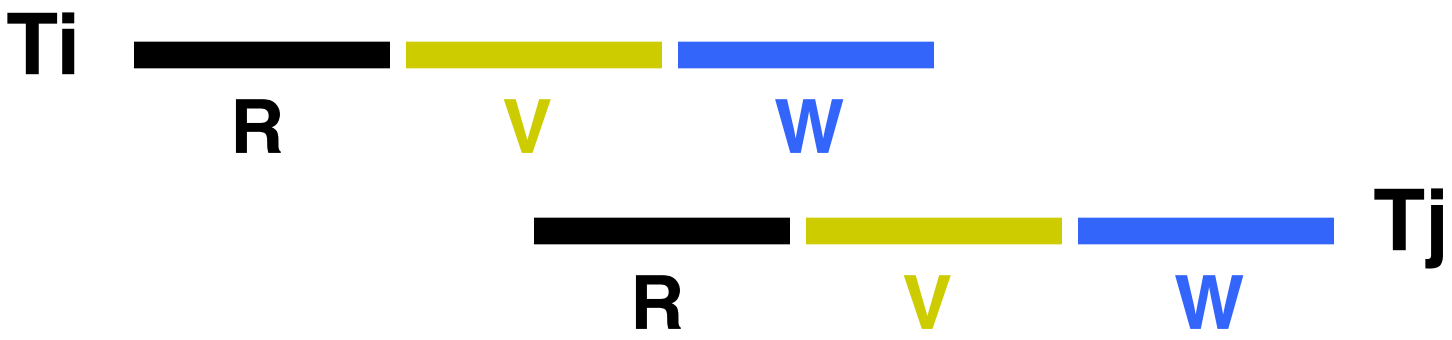
- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

Test 2 – No Write Phase Conflict

- For all i and j such that Ti < Tj, check that:

  Ti completes before Tj begins its Write phase

  and WriteSet(Ti) ∩ ReadSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Test 3 – Overlapping Write Phases

- For all i and j such that Ti < Tj, check that:

  Ti completes Read phase before Tj does **+**

  WriteSet(Ti) ∩ ReadSet(Tj) is empty **+**

  WriteSet(Ti) ∩ WriteSet(Tj) is empty.

Does Tj read dirty data? Does Ti overwrite Tj's writes?

Applying Tests 1, 2, &3

- To validate Xact T:

```
valid = true;
// S = set of Xacts that committed after Begin(T)
//    (above defn implements Test 1)
//The following is done in critical section
<foreach  Ts in S do {
   if (ReadSet(T) intersects WriteSet(Ts)) OR
       (WriteSet(T) intersects WriteSet(Ts))
         then valid = false;
   }>
   if valid then { install updates; // Write phase
                          Commit T }
              else Restart T
```

start of critical section

end of critical section

Applying Tests 1 & 2: Serial Validation

- To validate Xact T:

```
valid = true;
// S = set of Xacts that committed after Begin(T)
//    (above defn implements Test 1)
//The following is done in critical section
< foreach  Ts in S do {
   if ReadSet(T) intersects WriteSet(Ts)
         then valid = false;
   }
   if valid then { install updates; // Write phase
                        Commit T } >
              else Restart T
```

start of critical section

end of critical section

Comments on Serial Validation

- Applies Test 2, with T playing the role of Tj and each Xact in Ts (in turn) being Ti.
- Assignment of Xact id, validation, and the Write phase are inside a critical section!
  - Nothing else goes on concurrently.
  - So, no need to check for Test 3 --- can't happen.
  - If Write phase is long, major drawback.
- Optimization for Read-only Xacts:
  - Don't need critical section (because there is no Write phase).

Overheads in Optimistic CC

- Must record read/write activity in ReadSet and WriteSet per Xact.
  - Must create and destroy these sets as needed.
- Must check for conflicts during validation, and must make validated writes ``global''.
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- Optimistic CC restarts Xacts that fail validation.
  - Work done so far is wasted; requires clean-up.

Timestamp-Based Protocols

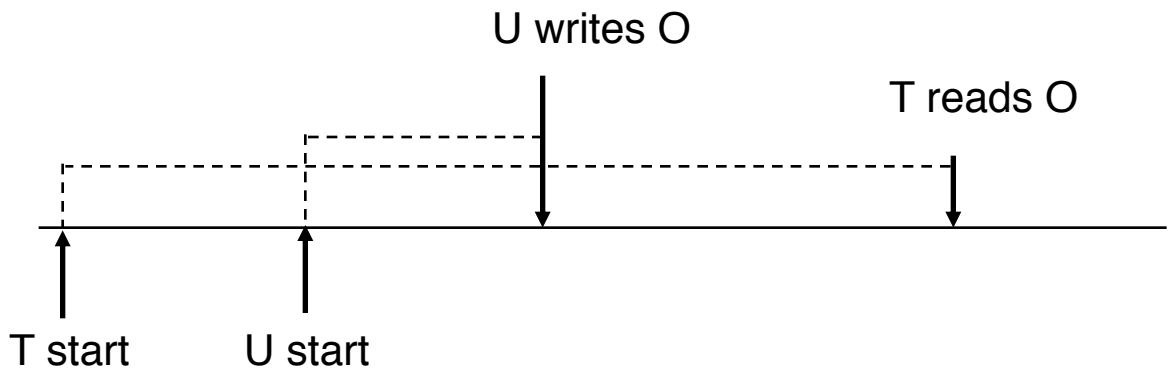
- Idea:
  - Decide in advance ordering of Xctions
  - Ensure concurrent schedule serializes to serial order decided
- Timestamps
  1. TS(T_i) is time T_i entered the system
  2. Data item timestamps:
     1. W-TS(O): Largest timestamp of any Xction that wrote O
     2. R-TS(O): Largest timestamp of any Xction that read O
- Timestamps -> serializability order

Timestamp CC

- **Idea:** If action p_i of Xact T_i conflicts with action q_j of Xact T_j, and $TS(T_i) < TS(T_j)$, then p_i must occur before q_j. Otherwise, restart violating Xact.

When Xact T wants to read Object O

- If TS(T) < W-TS(O), this violates timestamp order of T w.r.t. writer of O.
  - So, abort T and restart it with a new, larger TS. (If restarted with same TS, T will fail again!)
- If TS(T) > W-TS(O):
  - Allow T to read O.
  - Reset R-TS(O) to max(R-TS(O), TS(T))
- Change to R-TS(O) on reads must be written to disk (log)! This and restarts represent overheads.

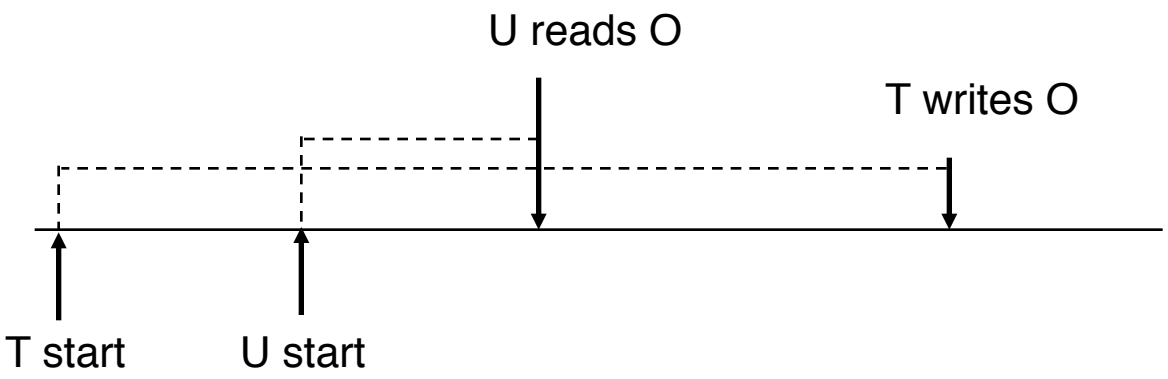


When Xact T wants to Write Object O

1) If TS(*T*) < R-TS(*O*), then the value of *O* that *T* is producing was needed previously, and the system assumed that that value would never be produced. **write** rejected, *T* is rolled back and restarts.

2) If TS(*T*) < W-TS(*O*), then *T* is attempting to write an obsolete value of *O*. Hence, this **write** operation is rejected, and *T* is rolled back.

3) Otherwise, the **write** operation is executed, and W-TS(*O*) is set to TS(*T*).



Another approach in 2) is to ignore the write and continue!!

Thomas Write Rule

Timestamp CC and Recoverability

- Unfortunately, unrecoverable schedules are allowed:

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

- Timestamp CC can be modified to allow only recoverable schedules:
  - Buffer all writes until writer commits (but update WTS(O) when the write is allowed.)
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.
- Similar to writers holding X locks until commit, but still not quite 2PL.

Locking in B+ Trees

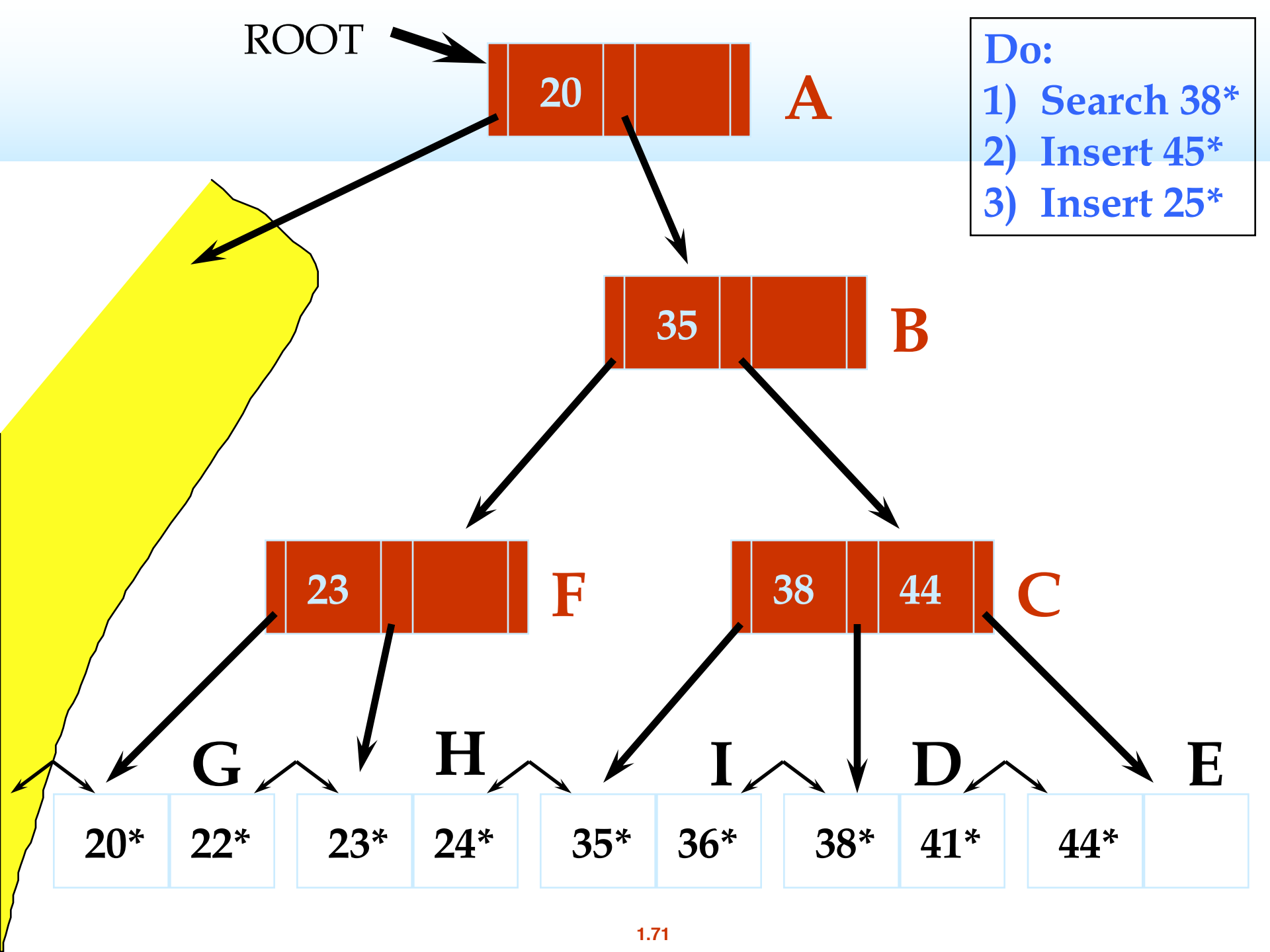
- How can we efficiently lock a particular leaf node?
  - Btw, don't confuse this with multiple granularity locking!
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
  - Root node (and many higher level nodes) becomes bottleneck 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 and keep them. Once child is locked, check if it is safe:
  - As you go, 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.

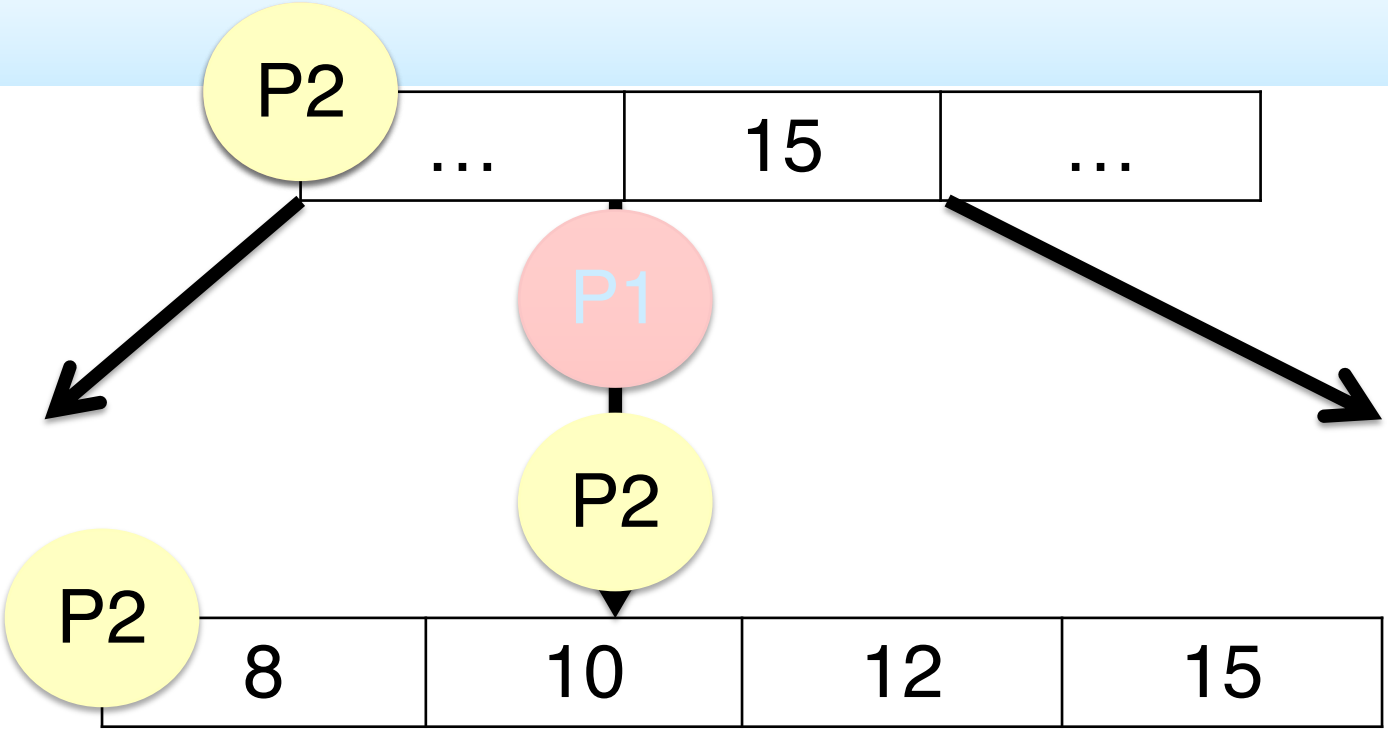

ROOT
20
A
35
B
23
F
38
44
C
G
H
I
D
E
20*
22*
23*
24*
35*
36*
38*
41*
44*
Do:
1) Search 38*
2) Insert 45*
3) Insert 25*

A Better Tree Locking Algorithm

- 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.

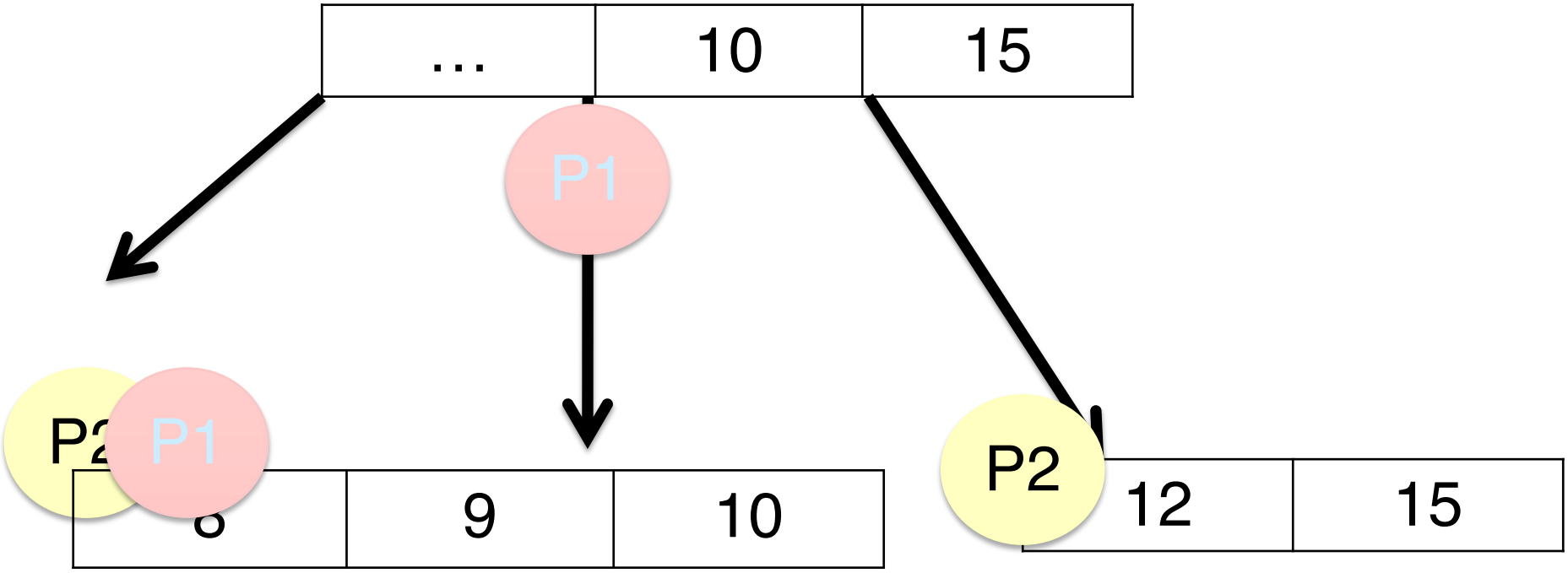
B^{link} - tree

Simple Approach



- P1 searches for 15
- P2 inserts 9

After the Insertion



- P1 searches for 15
- P2 inserts 9

P1 Finds no 15!

How could we fix this?

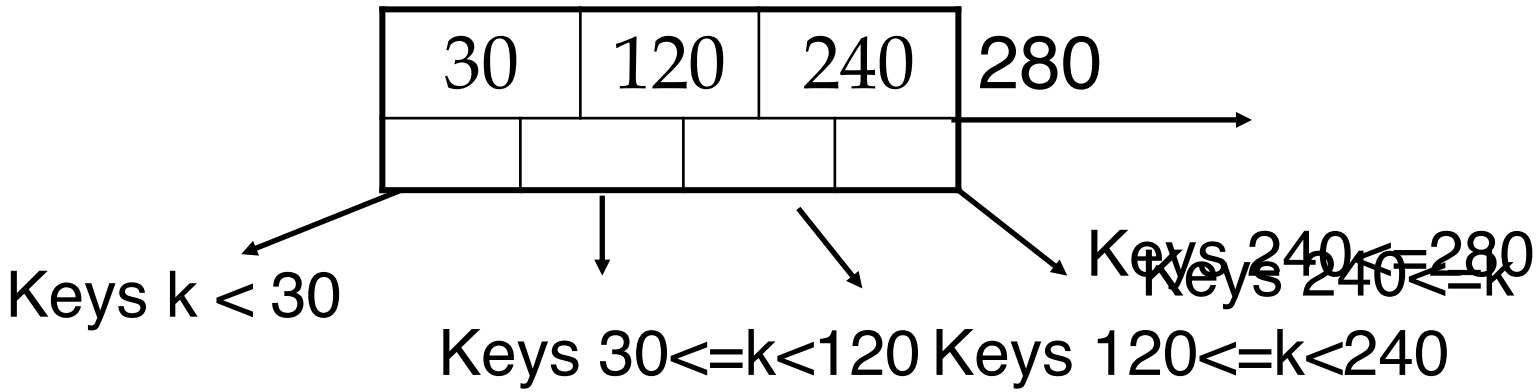
Two important Conventions

- Search for B-link trees root to leaf, left-to-right in nodes
- Insertions for B-link trees proceed bottom-up.

Internal Nodes

- Parameter d = the *degree*

Internal Node has
s >= d and <= 2d keys



Add right pointers.

We add a High key

Idea: If we get to this page, looking for 300. What can we conclude happened?

Valid Trees & Safe Nodes

- A node may not have a parent node, but it must have a left twin.
- We introduce the right links before the parent.
- A node is safe if it has [d,2d-1] pointers.

Scannode

scannode(u, A) : examine the tree node in A for value u and return the appropriate pointer from A.

Appropriate pointer may be the right pointer.

Searching for v

```
current = root;
A = get(current);
while (current is not a leaf) {
        current = scannode(v, A);
        A = get(current);}
while ((t = scannode(v,A)) == link pointer of A) {
        current = t;
        A = get(current);}
Return (v is in A) ? success :  failure;
```

Find the leaf w/ v

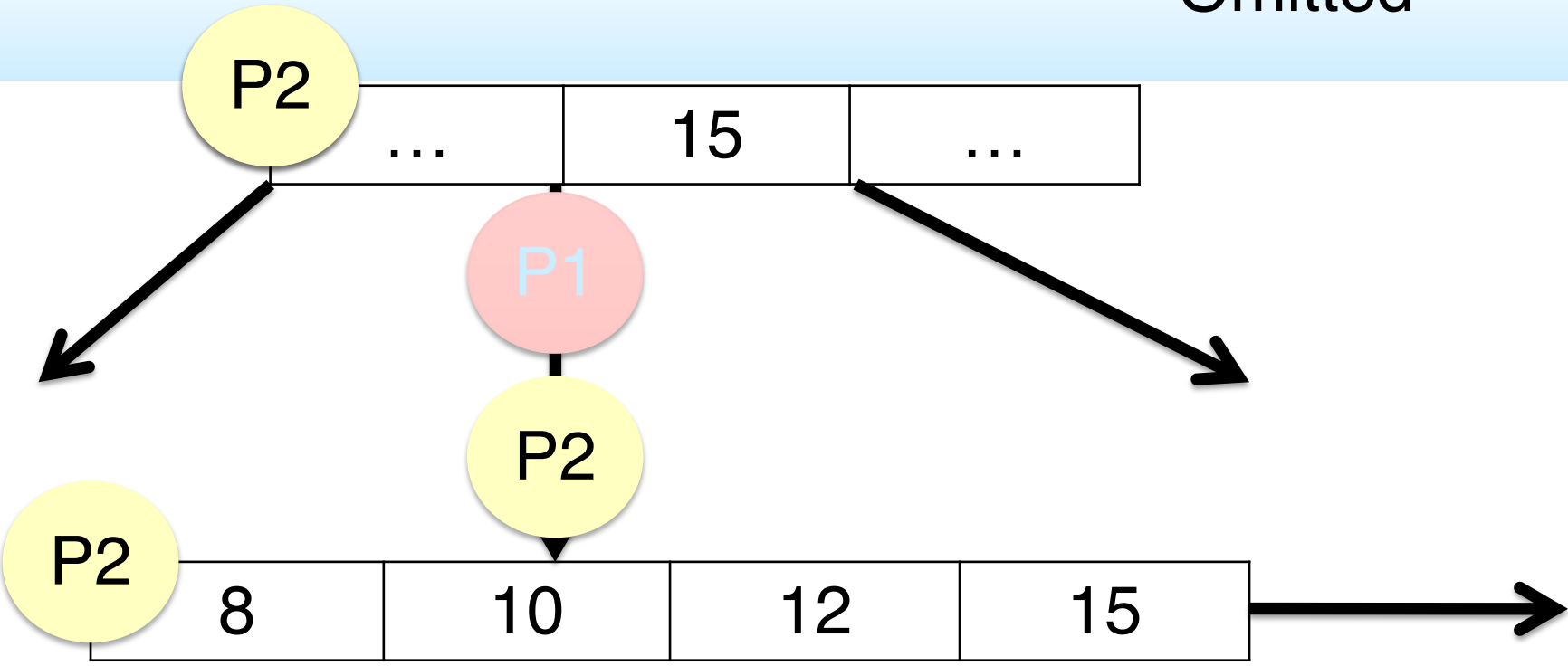
Find the leaf w/ v

Only modify scannode – No locking?!?

Insert

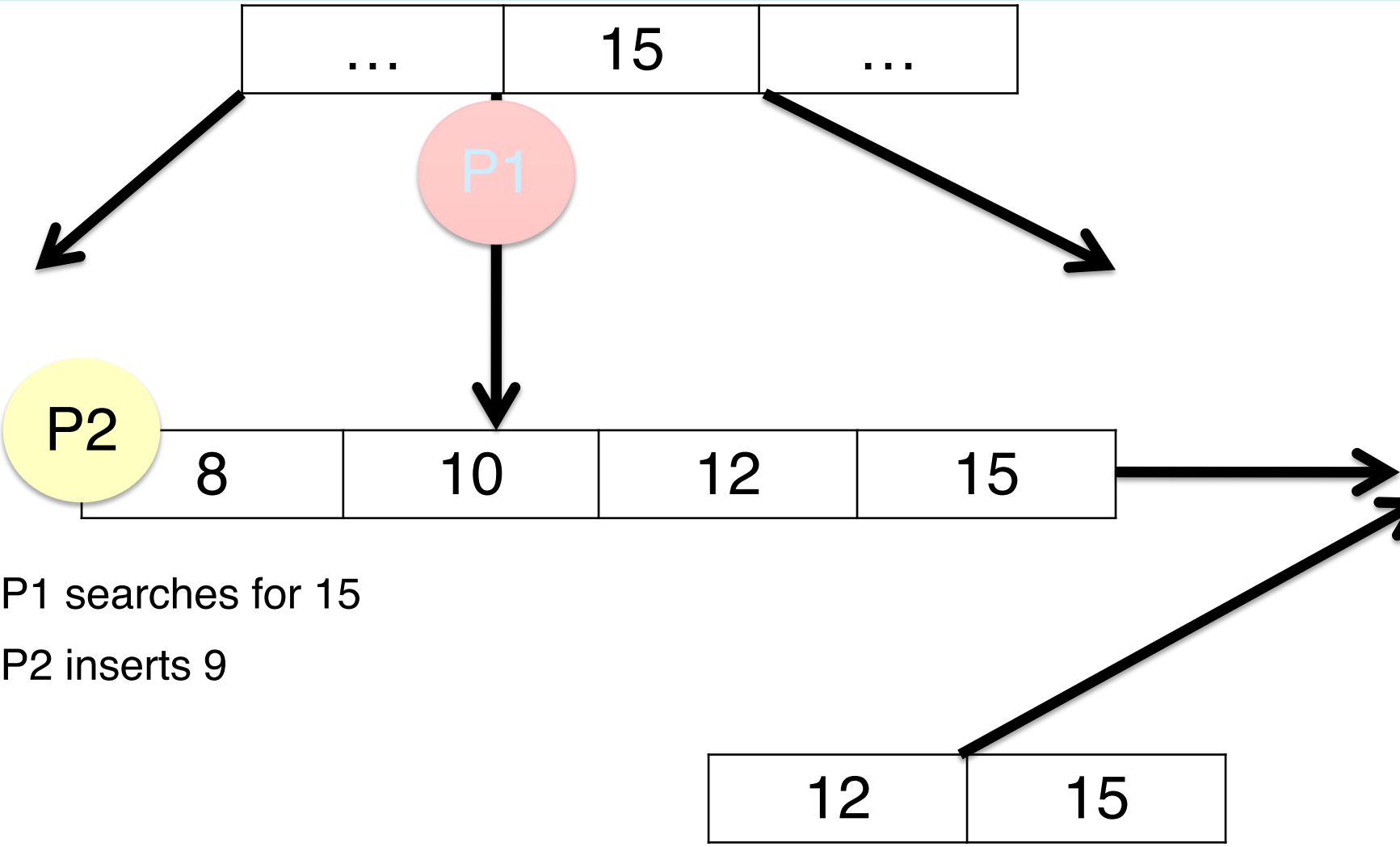
Revised Approach

High Key Omitted



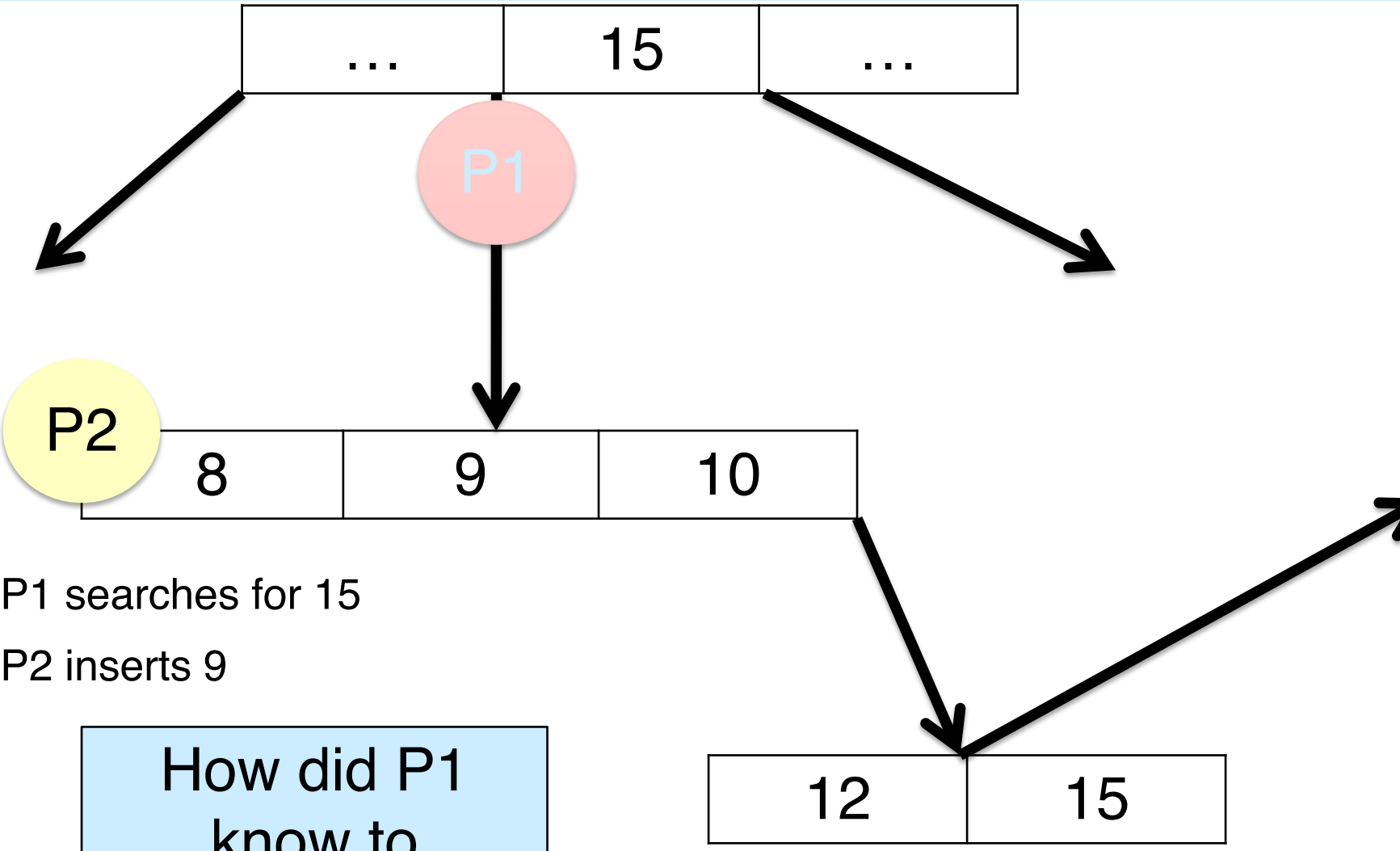
- P1 searches for 15
- P2 inserts 9

Revised Approach: Build new page



- P1 searches for 15
- P2 inserts 9

Revised Approach: Build new page



- P1 searches for 15
- P2 inserts 9

How did P1 know to continue?

Deadlock Free

Since the locks are placed by every process in a total order, there can be no deadlock. Why?

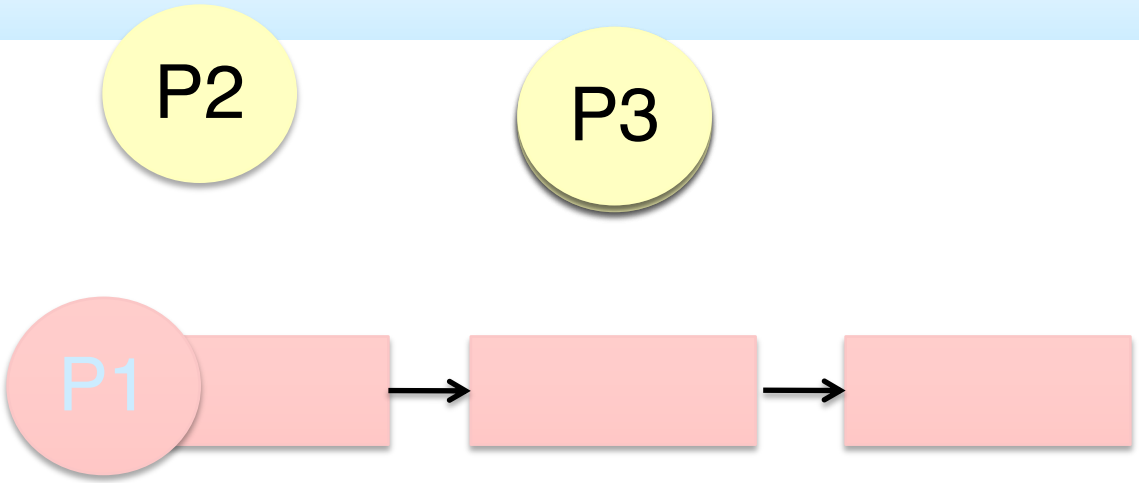
Is it possible to get the cycle:
T1(A) T2(B) T1(B) T2(A)?

Correct Interaction of Readers and Writers

Correct Interaction

Thm: Actions of an insertion process do not impair the correctness of the actions of other processes.

Livelock problem



Poor P1 never gets its value!
P1 is livelocked!

Further Reading

- Philip L. Lehman, S. Bing Yao:
  Efficient Locking for Concurrent Operations on B-Trees.
  ACM Trans. Database Syst. 6(4): 650-670 (1981)

- Recent HP Tech Report is great source (Graefe)

http://www.hpl.hp.com/techreports/2010/HPL-2010-9.pdf

Snapshot Isolation (SI)

- A multiversion concurrency control mechanism was described in SIGMOD ’95 by H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, P. O’Neil
  - Does not guarantee serializable execution!
- Supplied by Oracle DB, and PostgreSQL (before rel 9.1), for “Isolation Level Serializable”
- Available in Microsoft SQL Server 2005 as “Isolation Level Snapshot”

Snapshot Isolation (SI)

- Read of an item may not give current value
- Instead, use old versions (kept with timestamps) to find value that had been most recently committed at the time the txn started
  - Exception: if the txn has modified the item, use the value it wrote itself
- The transaction sees a "snapshot" of the database, at an earlier time
  - Intuition: this should be consistent, if the database was consistent before

First committer wins (FCW)

- T will not be allowed to commit a modification to an item if any other transaction has committed a changed value for that item since T's start (snapshot)
- Similar to optimistic CC, but only write-sets are checked
- T must hold write locks on modified items at time of commit, to install them.
  - In practice, commit-duration write locks may be set when writes execute.

Benefits of SI

- Reading is *never* blocked, and reads don't block writes
- Avoids common anomalies
  - No dirty read
  - No lost update
  - No inconsistent read
  - Set-based selects are repeatable (no phantoms)
- Matches common understanding of isolation: concurrent transactions are not aware of one another's changes
- On the downside – it turns out that it doesn't fully guarantee Serializablity (but Prof. Alan Fekete & team have fixed this in PostgreSQL 9.1+)

Summary

- Correctness criterion for isolation is “serializability”.
  - In practice, we use “conflict serializability”, which is somewhat more restrictive but easy to enforce.
- Two Phase Locking, and Strict 2PL: Locks directly implement the notions of conflict.
  - The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Must be careful if objects can be added to or removed from the database (“phantom problem”).
- Index locking common, affects performance significantly.
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).

Summary (Contd.)

- Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages);
  - should not be confused with tree index locking!
- Optimistic CC aims to allow progress when conflicts are rare or getting locks is expensive (e.g. distributed sys)
- Optimistic CC has its own overheads however; most real systems use locking or Snapshot Isolation.
- Snapshot Isolation is a practical approach that let's readers run without locks, by looking at (possibly) older snapshots.