Bazy danych 2022

Piotr Wieczorek

18 maja 2022

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)
- Pole nie ma wartości (student nie ma promotora (jeszcze))

Czym są NULLe?

- Pole ma jakąś wartość ale jej nie znamy (niepodpisany egzamin)
- Pole nie ma wartości (student nie ma promotora (jeszcze))
- Nie wiadomo, która z powyższych (może ma jakiegoś promotora, a może nie)

Czym są NULLe?

```
IF(OLD.text!=NEW.text) THEN        -- OLD.text<>NEW.text
          NEW.lasteditdate:=now();
          INSERT INTO commenthistory(commentid, creationdate, text)
             VALUES(OLD.id, OLD.lasteditdate, OLD.text);
```

Czym są NULLe?

```
IF(OLD.text!=NEW.text) THEN        -- OLD.text<>NEW.text
          NEW.lasteditdate:=now();
          INSERT INTO commenthistory(commentid, creationdate, text)
             VALUES(OLD.id, OLD.lasteditdate, OLD.text);
IF(OLD.text IS DISTINCT FROM NEW.text) THEN
          NEW.lasteditdate:=now();
          INSERT INTO commenthistory(commentid, creationdate, text)
             VALUES(OLD.id, OLD.lasteditdate, OLD.text);
```

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)
- `IS [ NOT ] NULL`

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)
- `IS [ NOT ] NULL`
- `a IS [ NOT ] DISTINCT FROM b`

Czym są NULLe?

- Operacje arytmetyka, porównania na NULLach - wynikiem NULL (UNKNOWN)
- `IS [ NOT ] NULL`
- `a IS [ NOT ] DISTINCT FROM b`
- Tabelki wartościowań:

a	*b*	*a* AND *b*	*a* OR *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT *a*
TRUE	FALSE

Czym są NULLe?

- `COUNT(*)` zlicza NULLe

Czym są NULLe?

- COUNT(*) zlicza NULLe
- COUNT(kol) nie zlicza NULLi

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę
- `SUM(kol)` zwraca NULL dla pustego zbioru krotek

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę
- `SUM(kol)` zwraca NULL dla pustego zbioru krotek
- `SELECT COALESCE(SUM(kol),0) FROM table WHERE 1=2`

Czym są NULLe?

- `COUNT(*)` zlicza NULLe
- `COUNT(kol)` nie zlicza NULLi
- `SUM(kol)` ignoruje NULLe, sumuje resztę
- `SUM(kol)` zwraca NULL dla pustego zbioru krotek
- `SELECT COALESCE(SUM(kol),0) FROM table WHERE 1=2`
- podobnie inne funkcje agregujące (za wyjątkiem `COUNT(*)` i `COUNT(kol)`, one zwracają 0)

Czym są NULLe?

Orders

order_id	title	price
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

Payments

cust_id	order_id
Cust1	Ord1
Cust2	Ord2

Customers

cust_id	name
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

Czym są NULLe?

Orders

order_id	title	price
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

Payments

cust_id	order_id
Cust1	Ord1
Cust2	Ord2

Customers

cust_id	name
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

Czym są NULLe?

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
      ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
      ( SELECT * FROM Orders O, Payments P
        WHERE  C.cust_id = P.cust_id
          AND  P.order_id = O.order_id )
```

Czym są NULLe?

ORDERS		
order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS	
cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS	
cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
      ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
      ( SELECT * FROM Orders O, Payments P
        WHERE  C.cust_id = P.cust_id
          AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY

Czym są NULLe?

ORDERS		
order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS	
cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS	
cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
      ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
      ( SELECT * FROM Orders O, Payments P
        WHERE  C.cust_id = P.cust_id
          AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY
- Co gdy w `Payments` wartość `Ord2` stanie się NULLem?

Czym są NULLe?

Orders

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

Payments

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

Customers

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
      ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
      ( SELECT * FROM Orders O, Payments P
        WHERE  C.cust_id = P.cust_id
          AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY
- Co gdy w `Payments` wartość `Ord2` stanie się NULLem?
- Unpaid orders -> EMPTY, Customers with no order -> `Cust2`

Czym są NULLe?

ORDERS

order_id	title	price
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	order_id
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	name
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

- Unpaid orders -> `Ord3`, Customers with no order -> EMPTY
- Co gdy w `Payments` wartość `Ord2` stanie się NULLem?
- Unpaid orders -> EMPTY, Customers with no order -> `Cust2`
- Więcej: P.Guagliardo, L. Libkin. *Correctness of SQL queries on databases with nulls.* SIGMOD Record (2017).

Transaction Management Overview

Chapter 16

Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A ***transaction*** is the DBMS's abstract view of a user program: a sequence of reads and writes.

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

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.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic.* That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

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

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

- Consider a possible interleaving (*schedule*):

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

- This is OK. But what about:

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

- 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:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) 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.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

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

- Unrepeatable Reads (RW Conflicts):

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

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

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

Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk ***before*** the changed page!
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - *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 and written to disk.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - *Consistent state*: Only the effects of commited Xacts seen.

Concurrency Control

Chapter 17

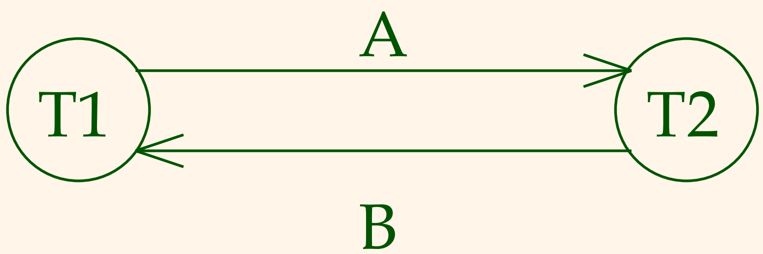
Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

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.

Dependency Graph

- *Dependency graph*: One node per Xact; edge from *Ti* to *Tj* if *Tj* reads/writes an object last written by *Ti*.
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

Review: Strict 2PL

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only schedules whose precedence graph is acyclic

Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

View 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

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

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: It Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits
- If a transaction re-starts, make sure it has its original timestamp

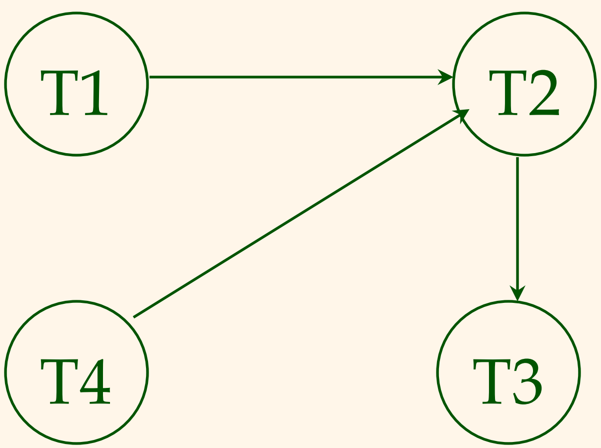
Deadlock Detection

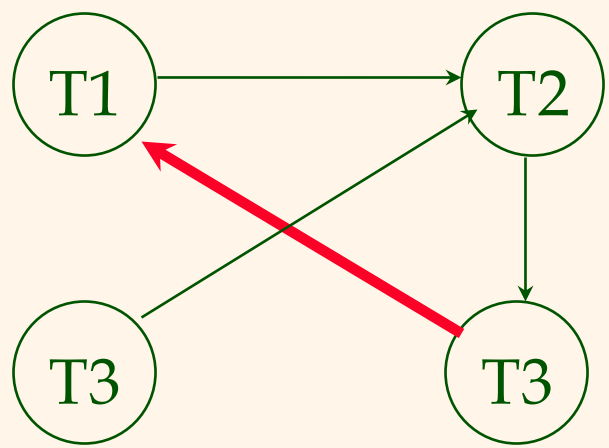
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

Deadlock Detection (Continued)

Example:

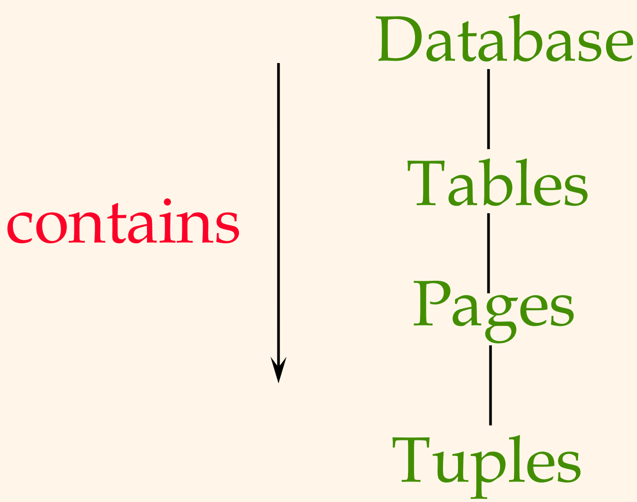
T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)



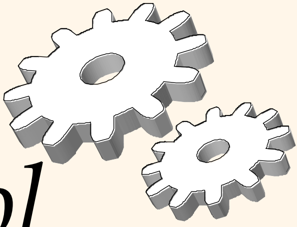


Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to 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 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.

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	√				

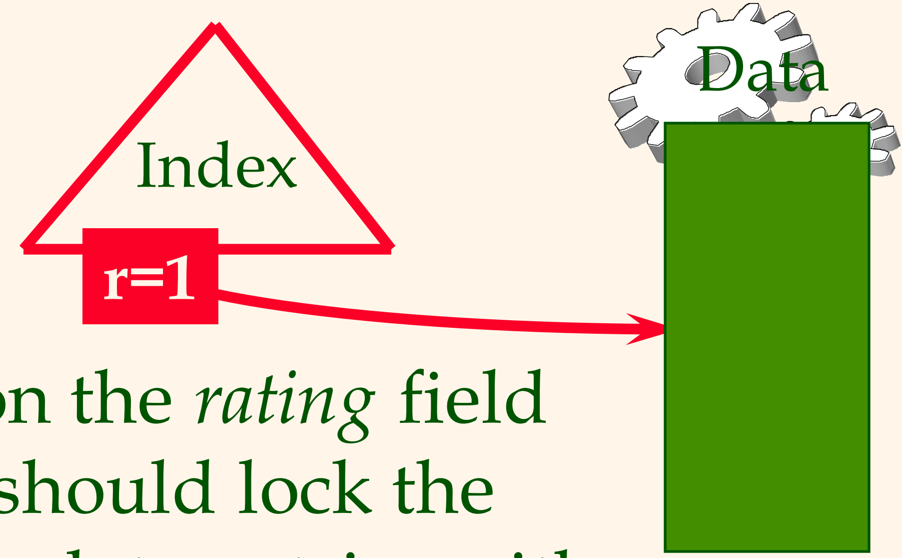
Dynamic Databases

- 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 where T1 is "correct"!

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

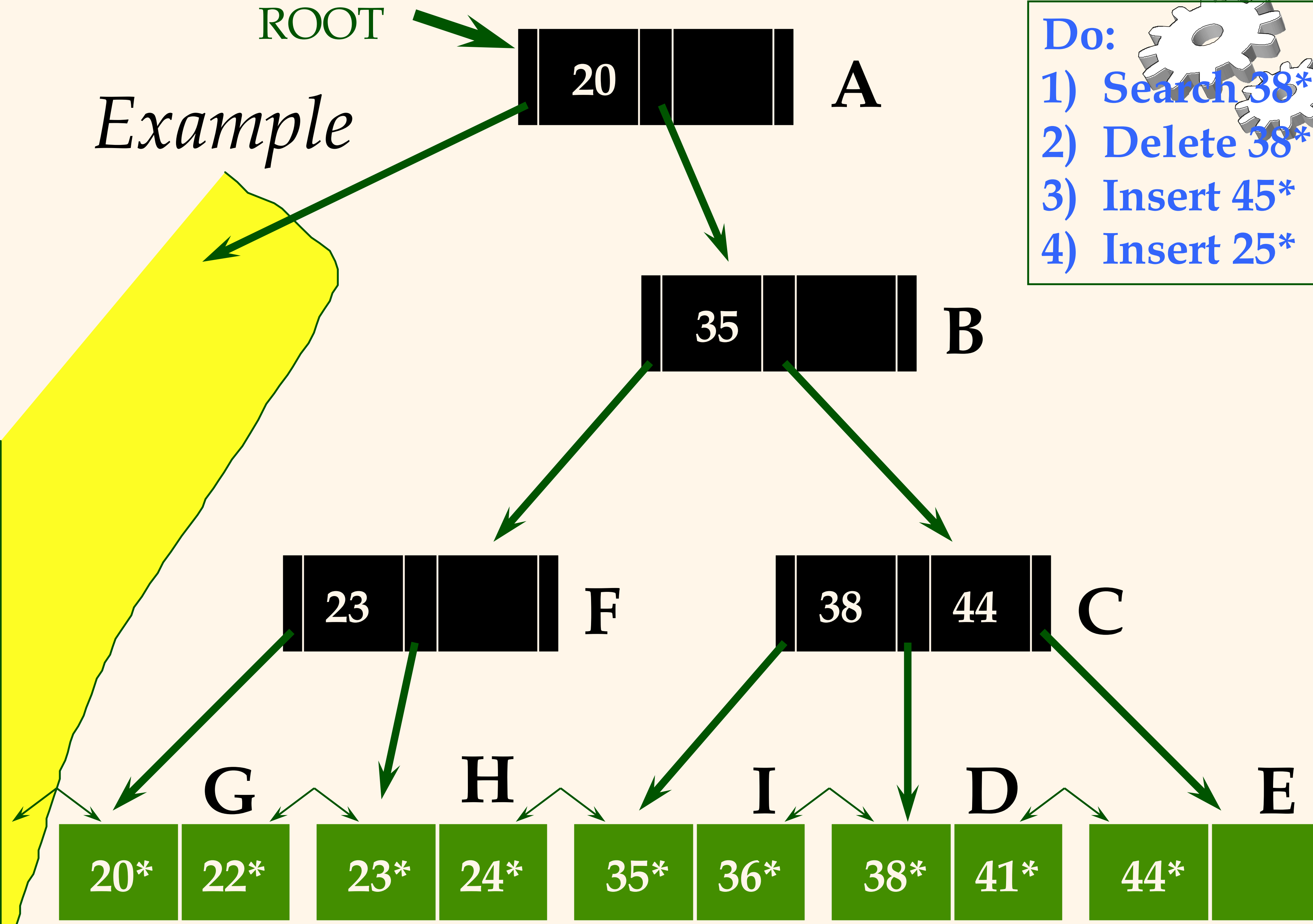
- Higher levels of the tree only direct searches for leaf pages.
- For inserts, a node on a path from root to modified leaf must be locked (in X mode, of course), only if a split can propagate up to it from the modified leaf. (Similar point holds w.r.t. deletes.)
- We can exploit these observations to 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.

Example

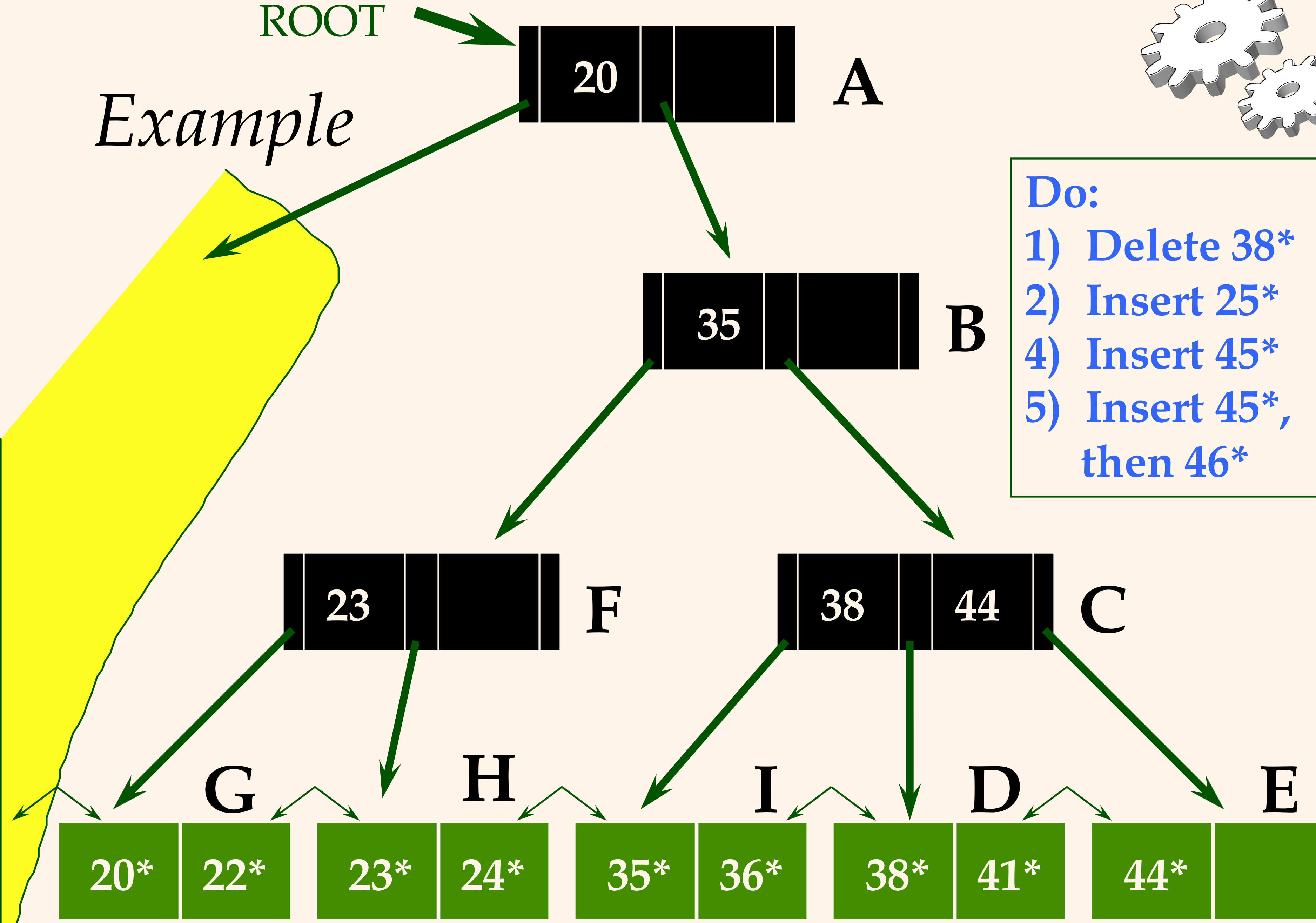
Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*



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.

Example



Do:

1) **Delete 38***
2) **Insert 25***
4) **Insert 45***
5) **Insert 45*, then 46***

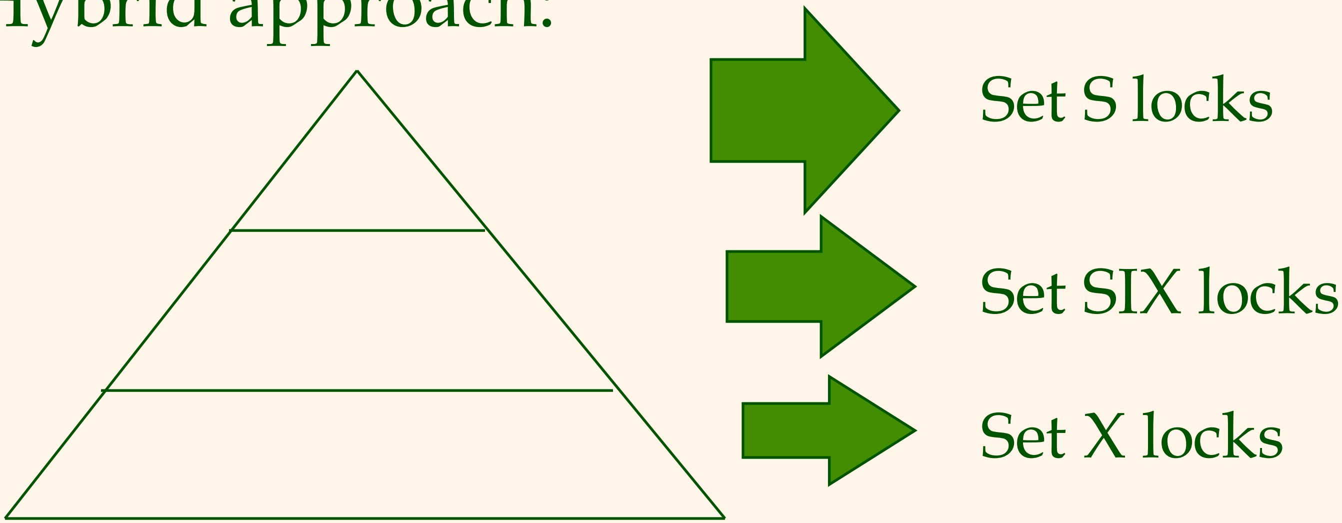
Even Better Algorithm

- 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:

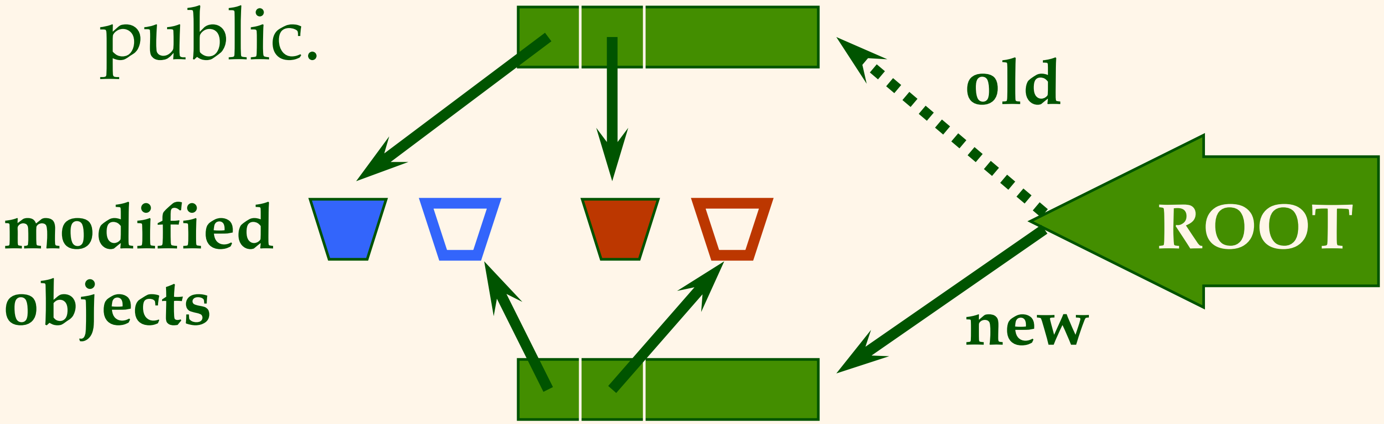


Optimistic CC (Kung-Robinson)

- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before Xacts 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

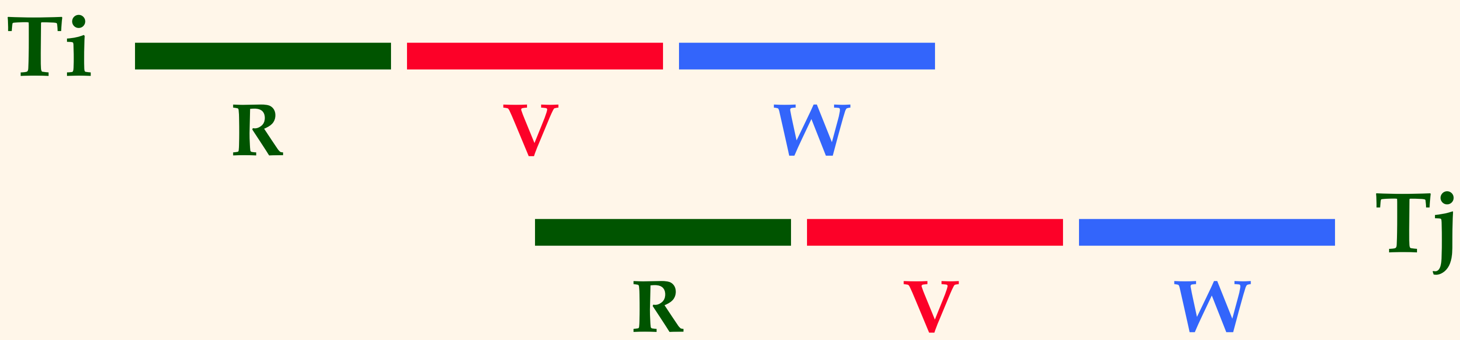
- Test conditions that are sufficient to ensure that no conflict occurred.
- Each Xact is assigned a numeric id.
  - Just use a **timestamp**.
- Xact ids assigned at end of READ phase, just before validation begins. (Why then?)
- ReadSet(Ti): Set of objects read by Xact Ti.
- WriteSet(Ti): Set of objects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

Test 1

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

Test 2

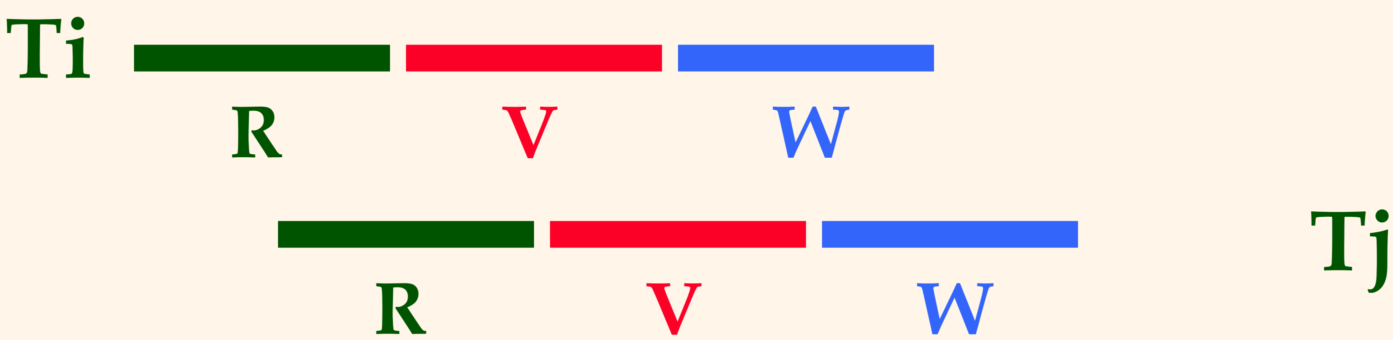
- For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase +
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty.



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

Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does +
  - WriteSet(Ti) $\cap$ ReadSet(Tj) is empty +
  - WriteSet(Ti) $\cap$ WriteSet(Tj) is empty.



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

Applying Tests 1 & 2: Serial Validation

- To validate Xact T:

```
valid = true;
// S = set of Xacts that committed after Begin(T)
< foreach  Ts in S do {
  if ReadSet(Ts) does not intersect WriteSet(Ts)
       then valid = false;
  }
  if valid then { install updates; // Write phase
                 Commit T } >
            else Restart T
```

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**!
  - I.e., Nothing else goes on concurrently.
  - If Write phase is long, major drawback.
- Optimization for Read-only Xacts:
  - Don't need critical section (because there is no Write phase).

Serial Validation (Contd.)

- Multistage serial validation: Validate in stages, at each stage validating T against a subset of the Xacts that committed after Begin(T).
  - Only last stage has to be inside critical section.
- Starvation: Run starving Xact in a critical section (!!)
- Space for WriteSets: To validate Tj, must have WriteSets for all Ti where Ti < Tj and Ti was active when Tj began. There may be many such Xacts, and we may run out of space.
  - Tj's validation fails if it requires a missing WriteSet.
  - No problem if Xact ids assigned at start of Read 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.

``Optimistic'' 2PL

- If desired, we can do the following:
  - Set S locks as usual.
  - Make changes to private copies of objects.
  - Obtain all X locks at end of Xact, make writes global, then release all locks.
- In contrast to Optimistic CC as in Kung-Robinson, this scheme results in Xacts being blocked, waiting for locks.
  - However, no validation phase, no restarts (modulo deadlocks).

Timestamp CC

- **Idea:** Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins:
  - If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Xact.

When Xact T wants to read Object O

- If TS(T) < WTS(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! Contrast use of timestamps in 2PL for ddlk prevention.)
- If TS(T) > WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to max(RTS(O), TS(T))
- Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.

When Xact T wants to Write Object O

- If TS(T) < RTS(O), this violates timestamp order of T w.r.t. writer of O; abort and restart T.
- If TS(T) < WTS(O), violates timestamp order of T w.r.t. writer of O.
  - **Thomas Write Rule: We can safely ignore such outdated writes; need not restart T! (T's write is effectively followed by another write, with no intervening reads.) Allows some serializable but non conflict serializable schedules:**
- Else, allow T to write O.

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

Timestamp CC and Recoverability

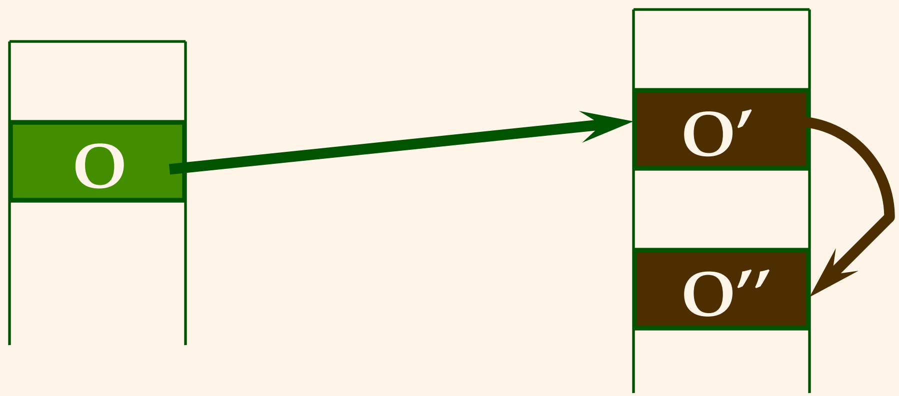
T1	T2
W(A)	
	R(A)
	W(B)
	Commit

- Unfortunately, unrecoverable schedules are allowed:
- 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.

Multiversion Timestamp CC

- **Idea:** Let writers make a "new" copy while readers use an appropriate "old" copy:

MAIN SEGMENT (Current versions of DB objects)



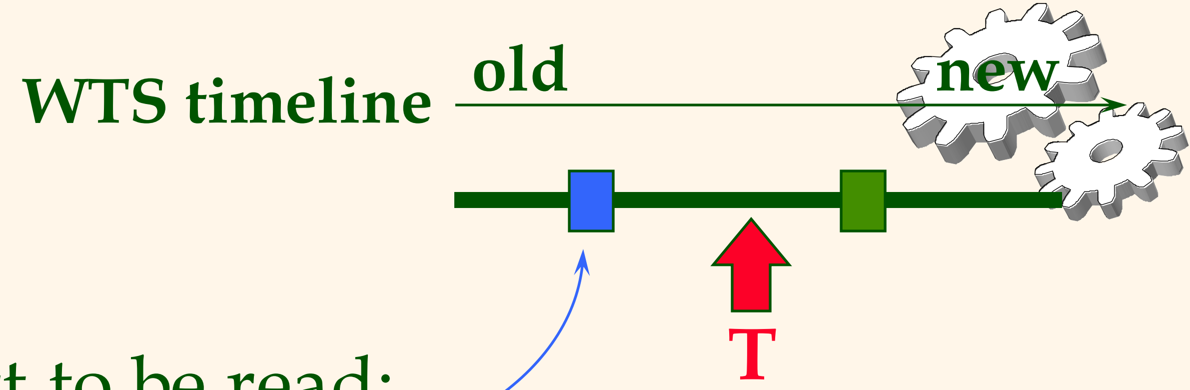
VERSION POOL (Older versions that may be useful for some active readers.)

- Readers are always allowed to proceed.
  - But may be blocked until writer commits.

Multiversion CC (Contd.)

- Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are "too old to be of interest".
- Each Xact is classified as Reader or Writer.
  - Writer *may* write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.

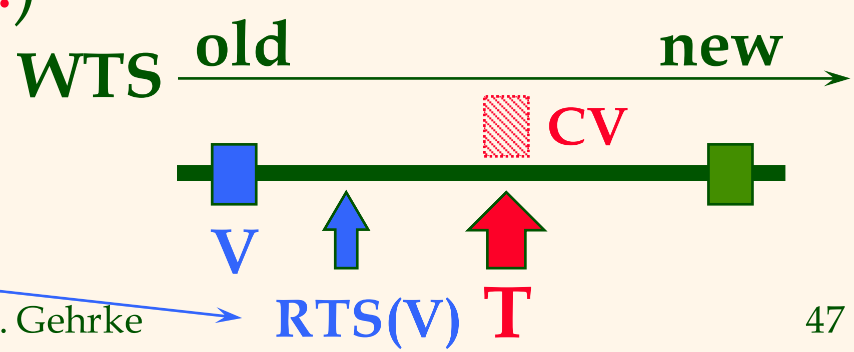
Reader Xact



- For each object to be read:
  - Finds **newest version** with WTS < TS(T). (Starts with current version in the main segment and chains backward through earlier versions.)
- Assuming that some version of every object exists from the beginning of time, Reader Xacts are never restarted.
  - However, might block until writer of the appropriate version commits.

Writer Xact

- To read an object, follows reader protocol.
- To write an object:
  - Finds **newest version V** s.t. WTS < TS(T).
  - If RTS(V) < TS(T), T makes a copy CV of V, with a pointer to V, with WTS(CV) = TS(T), RTS(CV) = TS(T). (Write is buffered until T commits; other Xacts can see TS values but can't read version CV.)
  - Else, reject write.



Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Naïve locking strategies may have the phantom problem

Summary (Contd.)

- Index locking is common, and affects performance significantly.
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).
- Tree-structured indexes:
  - Straightforward use of 2PL very inefficient.
  - Bayer-Schkolnick illustrates potential for improvement.
- In practice, better techniques now known; do record-level, rather than page-level 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 minimize CC overheads in an ``optimistic'' environment where reads are common and writes are rare.
- Optimistic CC has its own overheads however; most real systems use locking.
- SQL-92 provides different isolation levels that control the degree of concurrency

Summary (Contd.)

- Timestamp CC is another alternative to 2PL; allows some serializable schedules that 2PL does not (although converse is also true).
- Ensuring recoverability with Timestamp CC requires ability to block Xacts, which is similar to locking.
- Multiversion Timestamp CC is a variant which ensures that read-only Xacts are never restarted; they can always read a suitable older version. Additional overhead of version maintenance.