Concurrency Control

Goal of Concurrency Control

- Transactions should be executed so that it is *as though* they executed in some serial order
  - Also called Isolation or Serializability
- Weaker variants also possible
  - Lower "degrees of isolation"

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

- T1 transfers $100 from B's account to A's account
- T2 credits both accounts with 6% interest
- If submitted concurrently, net effect should be equivalent to Xacts running in some serial order
  - No guarantee that T1 "logically" occurs before T2 (or vice-versa) – but one of them is true

Solution 1

1) Get exclusive lock on entire database
2) Execute transaction
3) Release exclusive lock

- Similar to "critical sections" in operating systems
- Serializability guaranteed because execution is serial!

- Problems?

Solution 2

1) Get exclusive locks on *accessed* data items
2) Execute transaction
3) Release exclusive locks

- Greater concurrency

- Problems?

Solution 3

1) Get exclusive locks on data items that are *modified*; get shared locks on data items that are only *read*
2) Execute transaction
3) Release all locks

	S	X
S	Yes	No
X	No	No

- Greater concurrency
- Conservative Strict Two Phase Locking (2PL)

- Problems?

Solution 4

1) Get exclusive locks on data items that are modified and get shared locks on data items that are read
2) Execute transaction and release locks on objects no longer needed *during execution*

- ❖ Greater concurrency
- ❖ Conservative Two Phase Locking (2PL)

- ❖ Problems?

Solution 5

1) Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this *during execution* of transaction (as needed)
2) Release all locks

- ❖ Greater concurrency
- ❖ Strict Two Phase Locking (2PL)

- ❖ Problems?

Solution 6

1) Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this during execution of transaction (as needed)
2) Release locks on objects no longer needed during execution of transaction
3) ***Cannot acquire locks once any lock has been released***
   - ▪ *Hence two-phase (acquiring phase and releasing phase)*

- ❖ Greater concurrency
- ❖ Two Phase Locking (2PL)

- ❖ Problems?

Summary of Alternatives

- ❖ Conservative Strict 2PL
  - No deadlocks, no cascading aborts
  - But need to know objects a priori, when to release locks
- ❖ Conservative 2PL
  - No deadlocks, more concurrency than Conservative Strict 2PL
  - But need to know objects a priori, when to release locks, cascading aborts
- ❖ Strict 2PL
  - No cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
  - But deadlocks
- ❖ 2PL
  - Most concurrency, no need to know object a priori
  - But need to know when to release locks, cascading aborts, deadlocks

---

Method of Choice

- ❖ Strict 2PL
  - No cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
  - But deadlocks
- ❖ Reason for choice
  - Cannot know objects a priori, so no Conservative options
  - Thus only 2PL and Strict 2PL left
  - 2PL needs to know when to release locks (main problem)
    - Also has cascading aborts
  - Hence Strict 2PL
- ❖ Implication
  - Need to deal with deadlocks!

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

Outline

- ❖ Formal definition of serializability
- ❖ Deadlock prevention and detection
- ❖ Advanced locking techniques
- ❖ Lower degrees of isolation
- ❖ Concurrency control for index structures

Example

❖ Consider a possible interleaving (*schedule*):

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

❖ The DBMS's view of the 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 schedules is the same
  - ▪ The values read by transactions is the same in the schedules
    - • Assume no knowledge of transaction logic
- ❖ *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	

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

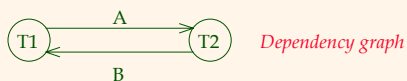❖ A schedule that is not conflict serializable:

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

```
            A
         ┌──────►
  T1 ◄───────────  T2      Dependency graph
            B
```

❖ 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

❖ Strict 2PL only allows conflict serializable schedule
  ▪ Dependency graph is always acyclic

Returning to Definition of Serializability

❖ A schedule S is serializable if there exists a serial order SO such that:
  ▪ The state of the database after S is the same as the state of the database after SO
  ▪ The values read by each transaction in S is the same as that returned by each transaction in SO
    • Database does not know anything about the internal structure of the transaction programs

❖ Under this definition, certain serializable executions are not conflict serializable!

Example

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

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

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

Outline

❖ Formal definition of serializability
❖ Deadlock prevention and detection
❖ Advanced locking techniques
❖ Lower degrees of isolation
❖ Concurrency control for index structures

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
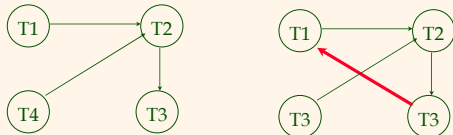
Deadlock Detection

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)
```

Outline

- ❖ Formal definition of serializability
- ❖ Deadlock prevention and detection
- ❖ Advanced locking techniques
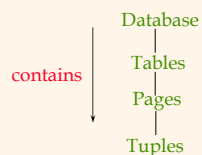- ❖ Lower degrees of isolation
- ❖ Concurrency control for index structures

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 <u>oldest</u> 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 <u>oldest</u> (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

Index

r=1
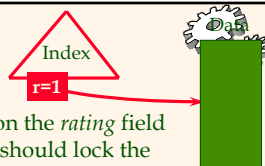
Data

- ❖ 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.

Outline

- Formal definition of serializability
- Deadlock prevention and detection
- Advanced locking techniques
- Lower degrees of isolation
- Concurrency control for index structures

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

Outline

- Formal definition of serializability
- Lower degrees of isolation
- Deadlock prevention and detection
- Advanced locking techniques
- Concurrency control for index structures

Locking in B+ Trees

- How can we enable "safe" concurrent access to index structures?
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- Problem?
  - 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.
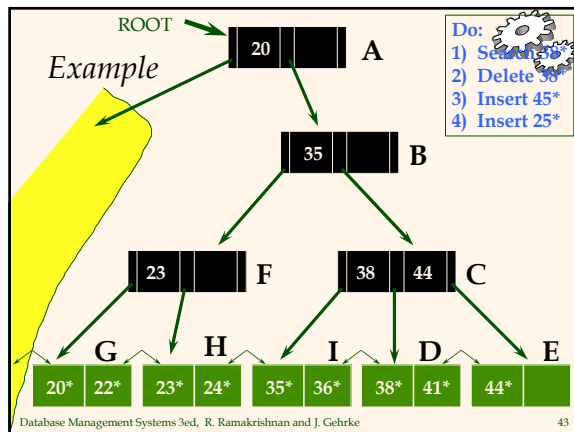 - Inserts: Node is not full.
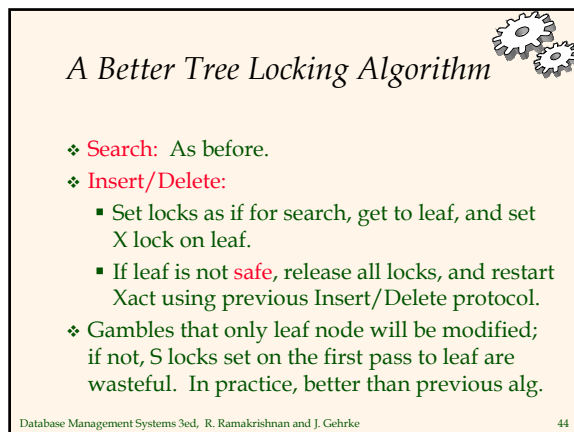 - Deletes: Node is not half-empty.

Example

20	**A**

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

35	**B**

23	**F**		**38**	**44**	**C**

G **H** **I** **D** **E**

| 20* | 22* | 23* | 24* | 35* | 36* | 38* | 41* | 44* | |

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.

Example

20	**A**

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

35	**B**

23	**F**		**38**	**44**	**C**

G **H** **I** **D** **E**

| 20* | 22* | 23* | 24* | 35* | 36* | 38* | 41* | 44* | |