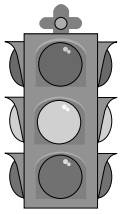
Concurrency Control

R &G - Chapter 19



Smile, it is the key that fits the lock of everybody's heart.

Anthony J. D'Angelo,
The College Blue Book

Administrivia

- **Homework 3 due next Sunday, 7pm**
- **No Office Hours today.**
- **Today: More Concurrency Control**
- **Tuesday: Logging/Crash Recovery w/Alexandra**
- **Next Thursday: Finishing Concurrency Control**

Review

- **We want DBMSs to have ACID properties**
- **These properties supported by:**
  - Transactions: unit of atomicity
  - Log: information to undo/redo transactions
  - Scheduler: limit reads/writes of Xactions to:
    - reduce anomalies
    - enhance concurency
- **Scheduling**
  - A serial execution of transactions is safe but slow
  - Try to find schedules equivalent to serial execution
  - One solution for serializable schedules is 2PL

Review: Transactions

- **Transactions either commit, or abort,**
  - no partial results
- **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.

Review: Anomalies

- **Reading Uncommitted Data ("WR", 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
```

- **Overwriting Uncommitted Data ("WW", lost update):**

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

Review: Anomalies

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

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

- **For example:**
  - T1 is transfers $100 from checking to savings, but aborts
  - T2 is supposed to add 6% interest

```
T1:  R(C), W(C-100),                R(S), W(S+100), Abort
T2:                R(C), W(C*1.06), C
```

  - Problem: the customer gets less interest!

Review: Anomalies

- **Unrepeatable Reads ("RW" Conflicts):**

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

- **For example,**
  - T1 transfers $100 from Checking to Savings
  - T2 gets Checking balance, withdraws $100

T1:	R(C)	W(C-100)	R(S)		W(S+100), C
T2:		R(C)		R(C)	W(C-100) C

  - Problem: Checking balance could go negative!

Review: Anomalies

- **Overwriting Uncommitted Data ("WW", lost update):**

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

- **For example,**
  - T1 transfers $100 from checking to money-market,
  - T2 transfers $100 from savings to money-market

T1:	R(C)	W(C-100)	R(M)	W(M+100)	
T2:		R(S)	W(S-100)	R(M)	W(M+100)

  - Problem: customer loses $100!

One more anomaly: Phantom Problem

- **If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL (on individual items) will not assure serializability:**
- **Consider T1 – "Find oldest sailor for each rating"**
  - 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 serial execution where T1's result could happen!**
  - Let's try it and see!

Review: Anomalies (cont.)

- **Some anomalies might be acceptable sometimes**
- **SQL 92 supports different "Isolation Levels" for a transaction (Lost Update not allowed at any 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

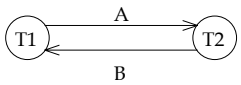
Concurrency Control - The big picture

- **Many ways to describe schedules**
  - Serial
  - Conflict Serializable
  - View Serializable
  - Strict
  - Avoid Cascading Abort
  - Recoverable
- **Sliding scale between anomalies and concurrency**
- **Precedence Graph a good way to understand schedules**

Dependency Graph a.k.a. Precedence Graph

- ***Dependency graph*:**
  - One node per Xact;
  - edge from *Ti* to *Tj* if *Tj* reads/writes an object last written by *Ti*.
- **Example:**

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

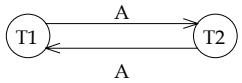


Dependency graph

Anomaly Dependency Graph

- **Unrepeatable Reads ("RW" Conflicts):**

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

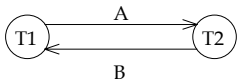


Dependency graph

Anomaly Dependency Graph

- **Overwriting Uncommitted Data ("WW", lost update):**

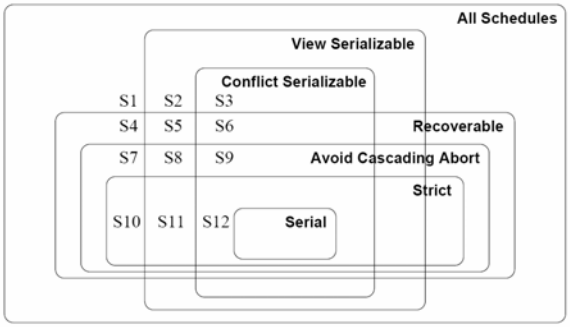
T1:	W(A),		W(B), C
T2:		W(A), W(B), C	



Dependency graph

Concurrency Control – A Big Picture

- **Tradeoffs between concurrency and anomalies**



Serial

- **A serial schedule runs transactions one-at-a-time**
- **It is guarrantied to be safe**
  - If each Xact is consistent, the serial schedule will be
- **It is inefficient, allowing no concurrency**

Conflict Serializable

- **Two actions *conflict* if they deal with the same object, and one or both are a Write.**
- **Two schedules are *conflict equivalent* if they order all conflicting actions the same**
- **A schedule is *conflict serializable* if it is conflict equivalent with a serial schedule.**
- ***Theorem*: Schedule is conflict serializable if and only if its dependency graph is acyclic**
- **Regular 2PL allows only conflict serializable schedules**

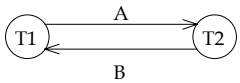
Conflict Serializable Schedules

- **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.**

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)

- **View serializability is "weaker" than conflict serializability!**
  - Every conflict serializable schedule is view serializable, but not vice versa!
  - I.e. admits more legal schedules

Recoverable

- **In a recoverable schedule, transactions commit only after all transactions whose changes they read commit.**
- **A recoverable schedule helps problem of reading uncommitted data, though you still have to do cascading abort.**
- **Not recoverable:**

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

- **Recoverable:**

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

Avoid Cascading Abort

- **A schedule that avoids cascading abort is one where transactions only read data written by committed transactions.**
- **These schedules are also recoverable.**
- **Strict 2PL avoids cascading aborts.**

Strict

- **A schedule is strict if all values written by a transaction T may not be read by any other transaction until T has committed.**
- **Strict schedules are recoverable, do not require cascading aborts.**
- **"Strict 2PL" is the strict version of 2PL**

Serializable

- **A schedule is serializable if the resulting database is equivalent to the result of some serial schedule.**
- **This is very hard to prove, hence conflict serializable, view serializable, etc.**
- **A serializable schedule is not necessarily recoverable**

App-Specific Serializability

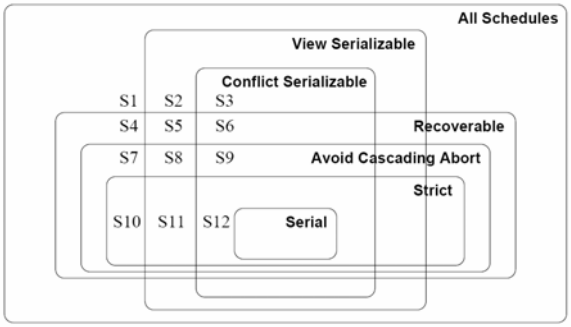
- **In some cases, application logic can deal with apparent conflicts**
  - E.g. when all writes commute
    - E.g. increment/decrement (a.k.a. "escrow transactions")

T1:	x=R(A), W(A=x+1),	z=R(A), W(z=z+1)
T2:	y=R(A), W(A=y-1)	

    - Note: doesn't work in some cases for (American) bank accounts
      - Account cannot go below $0.00!!
- **In general, this kind of app logic is not known to DBMS**
  - Only sees encapsulated R/W requests
  - But keep in mind that general serializability is "weaker" than even view serializability

Concurrency Control – The Big Picture

- **Tradeoffs between concurrency and anomalies**



Review: Strict 2PL

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

- ***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 a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- **Can result in Cascading Aborts!**
  - STRICT (!!) 2PL "Avoids Cascading Aborts" (ACA)

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**
  - requires latches ("semaphores"), which ensure that the process is not interrupted while managing lock table entries
  - see CS162 for implementations of semaphores
- **Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock**
  - Can cause deadlock problems

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 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 gets its original timestamp**
  - Why?

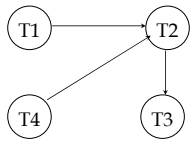
Deadlock Detection

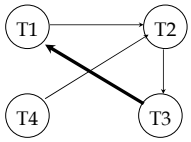
- **Create a waits-for graph:**
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- **Periodically check for cycles in the waits-for graph**

Deadlock Detection (Continued)

Example:

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





Deadlock Detection (cont.)

- **In practice, most systems do detection**
  - Experiments show that most waits-for cycles are length 2 or 3
  - Hence few transactions need to be aborted
  - Implementations can vary
    - Can construct the graph and periodically look for cycles
    - Can do a "time-out" scheme: if you've been waiting on a lock for a long time, assume you're deadlock and abort

Summary

- **Correctness criterion for isolation is "serializability".**
  - In practice, we use "conflict serializability", which is somewhat more restrictive but easy to enforce.
- **There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Locks directly implement the notions of conflict.**
  - The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.

Things to discuss next time

- **What should we lock?**
  - We assume tuples here, but that can be expensive!
  - If we do table locks, that's too conservative
  - *Multi-granularity* locking
- **Locking in indexes**
  - don't want to lock a B-tree root for a whole transaction!
  - actually do non-2PL "latches" in B-trees
- **CC w/out locking**
  - "optimistic" concurrency control
  - "timestamp" and multi-version concurrency control
  - locking usually better, though

In case we have time

- **The following is an interesting problem**
- **We will not discuss how to solve it, though!**

Dynamic Databases – The "Phantom" Problem

- **If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL (on individual items) will not assure serializability:**
- **Consider T1 – "Find oldest sailor for each rating"**
  - 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 serial execution where T1's result could happen!**
  - Let's try it and see!

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!**
  - e.g. table locks

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.**
  - too expensive!

Instead of predicate locking

- **Table scans lock entire tables**
- **Index lookups do "next-key" locking**
  - physical stand-in for a logical range!