Concurrency Control

Chapter 17

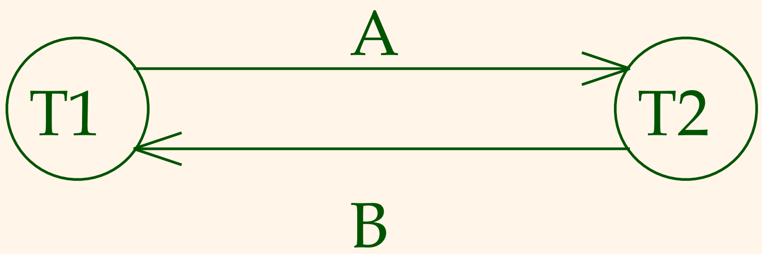
Conflict Serializable Schedules

- 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.

Two Transactions

T1

T2

A = 1000, B = 1000

1. Lock(X,A)
2. Read(A)
3. A = A + 100
4. Write(A)
5. Lock(X, B)
6. Unlock(A)
7. Read(B)
8. B = B - 100
9. Write(B)
10. Unlock(B)

1. Lock(X,A)
2. Read(A)
3. A = A*1.05
4. Write(A)
5. Lock(X,B)
6. Read(B)
7. B = B*1.05
8. Write(B)
9. Unlock(A)
10. Unlock(B)

T1 starts details

1. T1 Lock(X,A)
2. T1 Read(A)
3. T1 A = A + 100
4. T1 Write(A)
5. T1 Lock(X,B)
6. T1 Unlock(A), T2 Lock(X,A)
7. T1 Read)B), T2 Read(A)
8. T1 B = B - 100, T2 A = A*1.05
9. T1 Write(B), T2 Write(A)
10. T1 Unlock(B), T2 Lock(X,B)
11. T2 Read(B)
12. T2 B = B * 1.05
13. T2 Write(B)
14. T2 Unlock(A)
15. T2 Unlock(B)

- T2 then T1 strictly
- T1 (1-6), T2(1-4), T2(5,10)
- T1(7-10)

Two Transactions

T1

1. Lock(X,A)
2. Read(A)
3. A = A + 100
4. Write(A)
5. Lock(X, B)
6. Unlock(A)
7. Read(B)
8. B = B - 100
9. Write(B)
10. Unlock(B)

T2

1. Lock(X,A)
2. Read(A)
3. A = A*1.05
4. Write(A)
5. Lock(X,B)
6. Unlock(A)
7. Read(B)
8. B = B*1.05
9. Write(B)
10. Unlock(B)

A = 1000, B = 1000

T1 starts details

1. T1 Lock(X,A)
2. T1 Read(A)
3. T1 A = A + 100
4. T1 Write(A)
5. T1 Lock(X,B)
6. T1 Unlock(A), T2 Lock(X,A)
7. T1 Read)B), T2 Read(A)
8. T1 B = B - 100, T2 A = A*1.05
9. T1 Write(B), T2 Write(A)
10. T1 Unlock(B), T2 Lock(X,B)
11. T2 Read(B),
12. T2 B = B * 1.05
13. T2 Write(B)
14. T2 Unlock(A)
15. T2 Unlock(B)

T2 starts details

1. T2 Lock(X,A)
2. T2 Read(A)
3. T2 A = A * 1.05
4. T2 Write(A)
5. T2 Lock(X,B)
6. T2 Unlock(A), T1 Lock(X,A)
7. T2 Read)B), T1 Read(A)
8. T2 B = B*1.05, T1 A = A+100
9. T2 Write(B), T1 Write(A)
10. T2 Unlock(B), T1 Lock(X,B)
11. T1 Unlock(A)
12. T1 Read(B)
13. T1 B = B - 100
14. T1 Write(B)
15. T1 Unlock(B)

- T2 (1-6), T1(1-4), T1(5,10)
- T2(7-10)
- T1 (1-6), T2(1-4), T2(5,10)
- T1(7-10)

Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry (hash table on object id):
  - 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
- Lock downgrade: transaction that holds an exclusive lock can be downgraded to a shared lock

Lock Requests

- S lock request: if queue is empty and currently unlocked or locked in S - grant request
  - update entry - count++, mode = S
- X lock request: if object unlocked - grant request
  - update entry - count = 1, mode = X
- Otherwise - transaction waits in queue
- Commit/abort - release all locks

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention - one practical technique is to use Conservative (or Rigorous) 2PL, i.e. acquire all locks at the beginning of Xact and release them at end
  - Deadlock detection

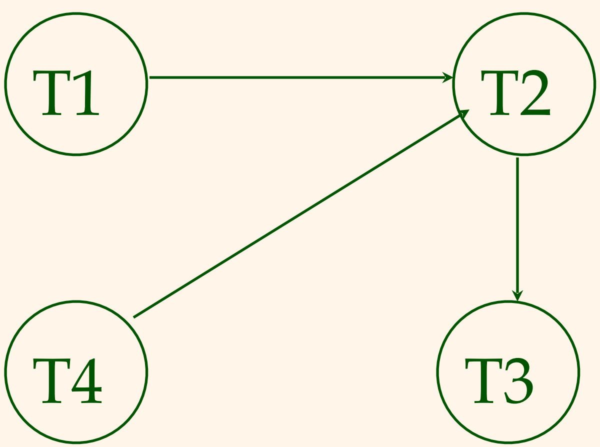
Deadlock Detection

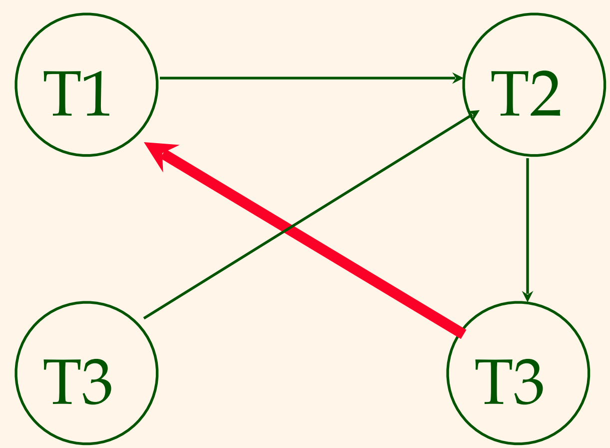
- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph

Deadlock Detection (Continued)

Example:

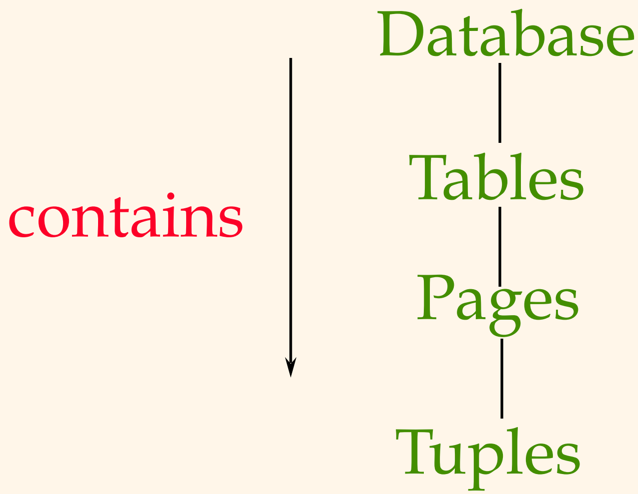
T1:	S(A), R(A),		S(B)				
T2:		X(B),W(B)			X(C)		
T3:				S(C), R(C)			X(A)
T4:						X(B)	





Multiple-Granularity Locks

- What granularity to lock (tuples vs. pages vs. tables).
- Intuitively: Use the one that would permit maximum concurrency
- Data "containers" are nested:



Dynamic Databases

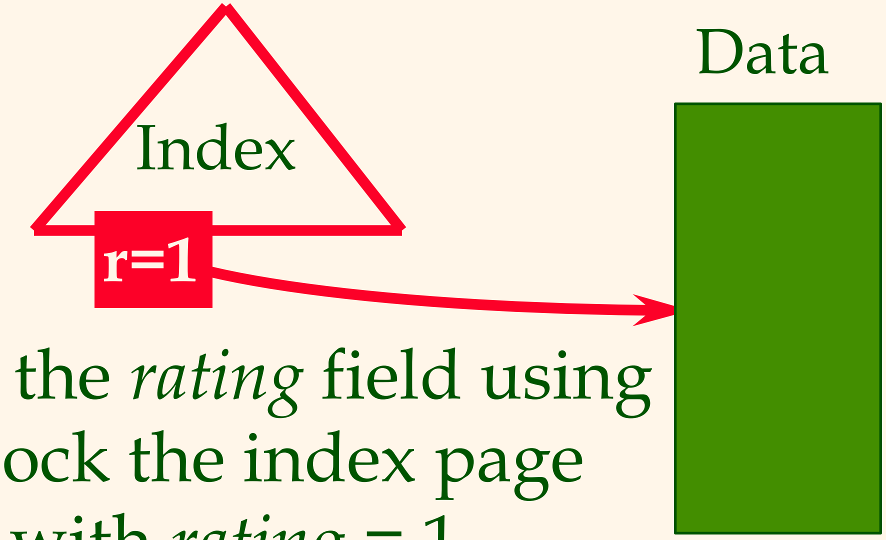
- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - T1 locks all pages containing sailor records with *rating* = 1, and finds <u>oldest</u> sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96 and commits
  - T1 returns with oldest sailor value of age = 71 (even though T2 "completed" before T1)
- No consistent DB state where T1 is "correct"!

插入删除与读写锁冲突。 即便有
2pl也破坏了consistency

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.) 解决办法
- 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.

Transaction Support in SQL-92

- Each transaction has an access mode (Read Only/Read Write), a diagnostics size (error size, not relevant), 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

Isolation levels and Locking

- Read Uncommitted - no shared locks acquired for reading, X locks as usual - Uncommon
- Read Committed - “short” read locks for reading i.e. shared locks released as soon as reading is done (i.e. not held till end), X locks as usual - Can be used for long running read mostly workload, e.g. data analysis
- Repeatable Read - strict 2PL/2PL - Uncommon
- Serializable - strict 2PL with handling of phantom phenomenon via predicate/index locking

Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL).
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- Naïve locking strategies may have the phantom problem
- In practice, better techniques do multi-level, rather than just page-level locking.