CS 245: Database System Principles

Notes 10: More TP

Hector Garcia-Molina
(Some modifications by Chris Olston)

Sections to Skim:

- Chapter 8:
  - skim 8.5
- Chapter 9:
  - skim 9.6, 9.8
- Chapter 10:
  - skim 10.4, 10.5, 10.6
- Chapter 11: none (read all sections)

Chapter 10 More on transaction processing

Topics:

- Cascading rollback, recoverable schedule
- Deadlocks
  - Prevention
  - Detection
- View serializability
- Distributed transactions
- Long transactions (nested, compensation)

Concurrency control & recovery

Example:

T_j	T_i
⋮	⋮
$W_j(A)$	
	$r_i(A)$
⋮	Commit T_i
⋮	
Abort T_j	⋮

☛ Need to "uncommit" Ti (Bad!)

- Schedule is conflict serializable
- $T_j \longrightarrow T_i$
- But not recoverable
  - I.e., not always possible to abort a transaction without affecting the status of some transaction that already committed

- Need to make "final" decision for each transaction:
  - **commit decision** - system guarantees transaction will or has completed, no matter what
  - **abort decision** - system guarantees transaction will or has been rolled back (has no effect)

To model this, two new actions:

- C_i - transaction T_i commits
- A_i - transaction T_i aborts

Note: in transactions, reads and writes precede commit or abort

⇨ If $C_i \in T_i$, then $r_i(A) < C_i$
$w_i(A) < C_i$

⇨ If $A_i \in T_i$, then $r_i(A) < A_i$
$w_i(A) < A_i$

- Also, one of C_i, A_i per transaction

Back to example:

T_j	T_i
⋮	⋮
$W_j(A)$	
⋮	$r_i(A)$
	⋮
	C_i ← can we commit here?

Definition

T_i reads from T_j in S ($T_j \Rightarrow_S T_i$) if

(1) $w_j(A) <_S r_i(A)$

(2) $a_j \not<_S r_i(A)$ ($\not<$: does not precede)

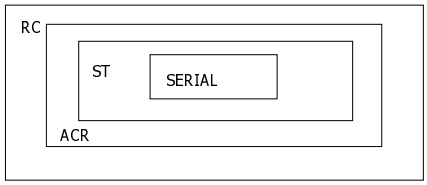
(3) If $w_j(A) <_S w_k(A) <_S r_i(A)$ then $a_k <_S r_i(A)$

Definition

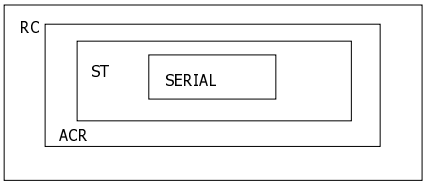
Schedule S is recoverable if whenever $T_j \Rightarrow_S T_i$ and $j \neq i$ and $C_i \in S$ then $C_j <_S C_i$

- S is recoverable if each transaction *commits* only after all transactions from which it read have committed.
- S avoids cascading rollback if each transaction may *read* only those values written by committed transactions.

- S is strict if each transaction may *read and write* only items previously written by committed transactions.



Where are serializable schedules?



Examples

- Recoverable:
  - $w_1(A)\ w_1(B)\ \ w_2(A)\ r_2(B)\ \ c_1\ c_2$
- Avoids Cascading Rollback:
  - $w_1(A)\ w_1(B)\ \ w_2(A)\ \ c_1\ \ r_2(B)\ \ c_2$

Assumes $w_2(A)$ is done without reading

- Strict:
  - $w_1(A)\ w_1(B)\ c_1\ \ w_2(A)\ r_2(B)\ \ c_2$

How to achieve schedules that avoid cascading rollback (and are thus also recoverable) ?

⇨ With 2PL, hold write locks until after commit (strict 2PL)

T_j	T_i
⋮	⋮
$W_j(A)$	⋮
⋮	⋮
C_j	⋮
$u_j(A)$	
⋮	$r_i(A)$

Avoids cascading rollback!

⇨ With validation, no change!

Deadlocks

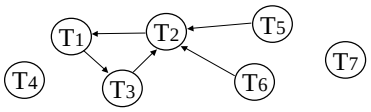
- Detection
  - Timeout
  - Wait-for graph
- Prevention
  - Resource ordering
  - Wait-die
  - Wound-wait

Timeout (Guess…)

- If transaction waits more than L sec., roll it back!
- Simple scheme
- Hard to select L
  - May wait too long before rolling back deadlocked transactions
  - May rollback transactions that were not deadlocked!

Real Deadlock Detection

- Build Wait-For graph
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim



Deadlock Prevention …

Prevention(1): Resource Ordering

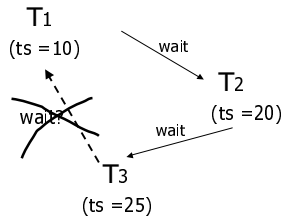
- Order all elements A_1, A_2, …, A_n
- A transaction T can lock A_i after A_j only if $i > j$

Problem : Ordered lock requests not realistic in most cases

Prevention(2): Wait-die

- Transactions given a timestamp when they arrive …. $ts(T_i)$
- T_i can only wait for T_j if $ts(T_i) < ts(T_j)$ ...else die

Example:



Prevention(3): Wound-wait

- Transactions given a timestamp when they arrive … ts(T_i)
- T_i wounds T_j if ts(T_i)< ts(T_j)
  else T_i waits

"Wound": T_j rolls back and gives lock to T_i

Locking

Locking

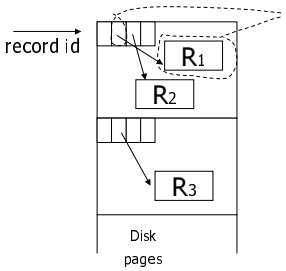
What are we really locking?

Example:

T_i ⋮
Read record r_1
⋮
Read record r_1
⋮
Modify record r_3
⋮

do record locking

But underneath:



If we lock all data involved in read of R1, we may prevent an update to R2 (which may require reorganization within block)

Solution: view DB at two levels

Top level: record actions
record locks
undo/redo actions — logical

e.g., Insert record(X,Y,Z)
Redo: insert(X,Y,Z)
Undo: delete

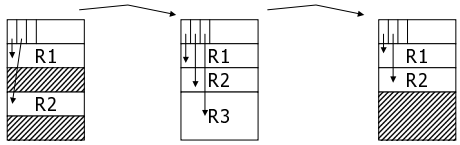
Low level: deal with physical details

latch page during action

(release at end of action)

Note: undo does not return physical DB to original state; only same logical state

e.g., Insert R3 Undo (delete R3)



We've looked at the low level guts of locking and concurrency control

Let's pop our heads out and look at things from the point of view of users

(Always a good exercise …)

User/Program commands

Lots of variations, but in general

- Begin_work
- Commit_work
- Rollback_work (Abort_work)

Transactions

User program (or SQL prompt):

⋮

Begin_work;

⋮

⋮

If results_ok, then commit work

else abort_work

Problem with Regular Transactions

Granularity of commit/abort is too coarse

Desirable behavior: see video games …

<save> <try to slay dragon> <save> <try to jump abyss> <save> …

Nested transactions

User program:

⋮

Begin_work;

Begin_work;

⋮

If results_ok, then commit work
else {abort_work; try something else…}

⋮

If results_ok, then commit work
else abort_work

Parallel Nested Transactions

T_1: begin-work

⋮

parallel:

T_{11}: begin_work

⋮

commit_work

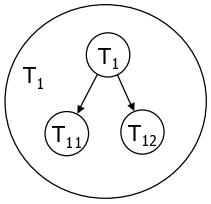
T_{12}: begin_work

⋮

commit_work

⋮

commit_work



Another Problem with Transactions

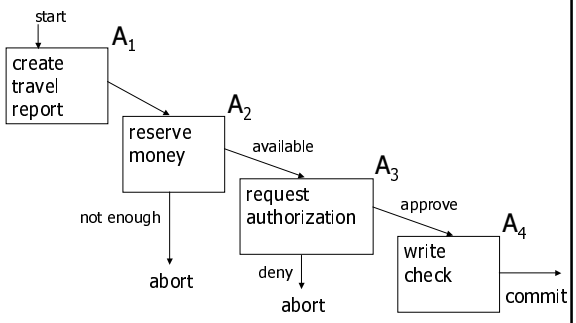
- Some transactions take a long time!
  - data analysis / mining
  - CAD systems
  - workflow
- Want to avoid locking out other transactions altogether
- At same time, avoid complete anarchy

Idea #1: Different Levels of Isolation

- For some long-running analysis or mining queries, can tolerate some inconsistencies
- For such transactions, do not enforce 2PL (i.e., permit unlock then lock)
- Permits other transactions to modify the data while the analysis query is running
- Lose guarantee of "repeatable read"

What about long-running transactions that modify the database?

Workflow example: expense report



Compensating Transactions

- For every action A, an action A^{-1} that "undoes" the effects of A, regardless of intervening actions
- Formally: $A\alpha A^{-1} \equiv \alpha$
- A_1^{-1} = delete travel report
- A_2^{-1} = put money back in account
- A_3^{-1} = delete authorization form
- A_4^{-1} = cancel check (never necessary)

Semantic atomicity:

execute one of

- A_1, A_2, A_3, A_4
- $A_1, A_2, A_3, A_3^{-1}, A_2^{-1}, A_1^{-1}$
- $A_1, A_2, A_2^{-1}, A_1^{-1}$
- A_1, A_1^{-1}
- nothing

called a saga

Careful!

- Compensating actions are tricky
- Remember requirement: $A\alpha A^{-1} \equiv \alpha$
- Ex: start with an account worth $2000
  A = add $1000; A^{-1} = deduct $1000
  C = deduct $2500

  Constraint: balance can't be negative

  ACA^{-1} → A^{-1} fails
  C → C fails

  $A\alpha A^{-1} \not\equiv \alpha$

Summary

- Cascading rollback
  Recoverable schedule
- Deadlock
  - Prevention
  - Detection
- Logical vs. physical locking
- Nested transactions
- Long running transactions