COS 597A:
Principles of
Database and Information Systems

Transactions and Concurrency Control

Transactions

- Unit of update/change
  - Viewed as indivisible
  - Database can be inconsistent during transaction
    - Add to relations with mutual foreign keys
    - Constraints on values
      - Debit of bank savings + credit of bank checking
  - Commit transaction/ Abort transaction
    - Aborts by User
    - Aborts by Error

Consistency

- Satisfies declared integrity constraints
- Satisfies semantics of correct execution of actions
  - Example: tuple not specified for deletion is still there after DELETE is executed

Concurrency

- Must be able to execute multiple transactions on DB together
  - Multiple users
    - Reservations, billing, banking, …
  - Long transactions
    - Reports, analysis, …
- Interleave transactions
- Each committed transaction must leave DB in consistent state
- Each aborted transaction must leave DB in state as if it never happened

ACID

Properties of transactions:

- **A**tomicity: all operations of a transaction are complete at commitment or none are
- **C**onsistency: each transaction in isolation leaves database in consistent state
- **I**solation: each transaction "unaware" of other transactions executing concurrently
- **D**urability: changes to database made by committed transactions persist even if system fails.

Database Management System must insure these

Modeling transactions

- Only reads and writes to DB tables relevant
- Consider actions READ, WRITE, COMMIT, ABORT
- How interleave these actions correctly?
  - Actions of different transactions can interact
- Around these actions a transaction does local computation: not affect DB
  - Example: comparison for query evaluation

Example

Transaction T1: debit savings; credit checking
Transaction T2: get checking balance; get savings balance

T1: debit savings   credit cking   BAD
T2:   bal. chking?   bal. savings?
→ time →

Transaction T1: debit savings; credit checking
Transaction T3: get savings balance; get checking balance

T1: debit savings   credit cking   GOOD
T3:   bal. saving?   bal. chking?
→ time →

Read/Write diagrams

→ time →

T1: R(V) W(V)   R(K)W(K)   C
T2:   R(K)   R(V)   C   X
T3:   R(V)   R(K)   C   √
T4:   R(K)R(V)C   X

R(object): read the DB object
W(object): write the DB object
C: transaction commits
V represents savings account
K represents checking account

Equivalence of schedules

Two schedule are equivalent if:

For any starting state of the DB for both schedules

The effect of executing the 1st schedule is identical to the effect of executing the 2nd schedule

Effect refers to the state of the DB as well as other results (e.g. a nasty letter that you are overdrawn)

Serializability

- Serial schedule: schedule for a set of transactions that does not interleave actions of different transactions
- A schedule is serializable if it is equivalent to some serial schedule for the same set of transactions

Conflict Serializable

- Conflicting actions by different transactions
  - Read and write to same DB object
  - Two writes to the same DB object
- Only non-conflicting actions to the same DB object
  - Two reads

A schedule is conflict serializable if the non-conflicting actions of the schedule can be reordered to get a serial schedule

– Strong condition!

Our Examples

→ time →

T1: R(V) W(V)   R(K)W(K)   C
T2:   R(K)   R(V)   C
           ←?→   ←?→
T3:   R(V)   R(K)   C
           ←?→   ←?→

Our Examples

time →

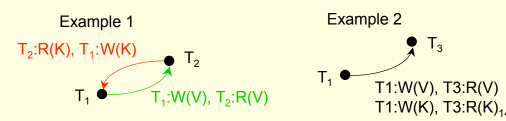
T1: R(V) W(V) R(K)W(K) C
T2: R(K) R(V) C X

T3: R(V) R(K) C √

Precedence Graph

- Each node represents a transaction T_i
- Edge from T_i to T_k if some action of T_i precedes and conflicts with an action of T_k

THEOREM: A schedule is conflict serializable if and only if the precedence graph for the schedule is acyclic



Locking

- Locks maintained by transaction manager
- Transaction requests lock
- Manager grants/denies lock
- Lock types:
  - Shared: need to have before read object
  - Exclusive: need to have before write object
- Object locked?
  - Different levels granularity
    - Tables and indexes
    - expense

Locking protocols

- Strict 2-phase locking:
  - Transaction requests lock at any time before action
  - Transaction releases locks when commits
- 2-phase locking (not strict)
  - Transaction requests lock at any time before action
  - Transaction releases locks at any time, BUT cannot request additional locks once released ***any*** lock
    - Can release before commit but must have all locks ever need when release 1st
- Strict 2-phase locking satisfies 2-phase locking constraints

Theorem

- 2 phase locking (2PL) allows only schedule with acyclic precedents graph

=>

- 2 phase locking allows only conflict serializable schedules

- Corollary: Strict 2-phase locking allows only conflict serializable schedules

Locking for our examples

T1: **S(V)** R(V) **X(V)** W(V) **S(K)** R(K) **X(K)** W(K) C
T2: **S(K)**R(K) ? R(V) C X

T2 can't get S(V) until T1 releases X(V)
BUT T1 can't release X(V) until gets X(K)
and T1 can't get X(K) until T2 releases S(K)
and T2 can't release S(K) until gets S(V)

T1: **S(V)**R(V)**X(V)**W(V)**X(K)**↑L(V) R(K) W(K)↑L(K) C
T3: **S(V)**R(V) **S(V)**R(K) C √

T1 can get X(K) in anticipation of writing K, then can release V

S(A): acquire shared lock on A **X(A)**: acquire exclusive lock on A
↑L(A) release all locks on A assume ↑L(any held lock) on commit

Serializable versus *conflict serializable*

- Are serializable schedules that are not conflict serializable

  T1: W(A) W(A)
  T2: W(A)

  Same result as
  T1: W(A) W(A)
  T2: W(A)

  W(A) not depend on R(A) - called ***blind write***

- conflict serializable stricter but easy to achieve

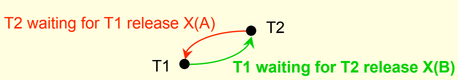
View serializable

- two schedules are ***view equivalent***
  - Informally, can't distinguish results of schedules:
    - transactions read same values of each object
    - last transaction to write each object same
  - Formally, each of the following must occur in $sched_1$ iff it occurs in $sched_2$
    - the initial value of an object A is read by T_i
    - T_j reads value that T_k writes
    - T_f executes the final write of an object A
- A schedule is ***view serializable*** if it is view equivalent to a serial schedule

Deadlock

- Transaction doesn't get lock ⇒ waits
  - transaction schedule: sequence of lock requests, lock releases, reads & writes
- deadlock: cycles of waiting

  T1 gets exclusive lock for object A
  T2 gets exclusive lock for object B
  T1 requests exclusive lock for object B
  T2 requests exclusive lock for object A



Deadlock prevention I

By way handle not getting requested lock

- One way: give priorities to transactions
  - based on time stamp
- Protocol to decide what happens when T_w wants lock & T_h holds lock:

  Wait-die: if priority(T_w) > priority(T_h), T_w waits
  otherwise T_w aborts
  Wound-wait: if priority(T_w) > priority(T_h), T_h aborts
  otherwise T_w waits
- For either, argue no cycle in "waiting for" graph
- Starvation?

Deadlock prevention II

- Change locking protocol
- Conservative two-phase locking:

  transaction acquires all locks ever needs
  at beginning of execution
  or waits with no locks
  - no transaction waiting on blocked transaction

Deadlock detection

- construct "waiting for" graph periodically
  & check for cycle
- must abort transaction to break cycle
  - how choose which?
    - last edge added? know?
  - heuristics

Aborting

- Why transactions abort?
  - Deadlock avoidance
  - System error
  - user command
- Dependent transactions could be forced to abort too:
  1. T_i aborts
  2. T_k read what T_i wrote

  =>

  3. T_k must abort (re-execute) EVEN IF T_k has committed!
     - What does "COMMIT" mean?

25

Cascaded aborts

2PH:

T_i: W(V) ↑L(V) … ABORT

T_k: → time R(V) COMMIT →

Strict 2PH:

- T_i releases locks and commits as atomic action
- Eliminates above problem

Choice of restrictions for conflicts :

- Strict: T_k does not read or write until T_i commits
- Avoid cascaded abort: T_k does not read until T_i commits
- Recoverable: T_k only commits after T_i commits
  - CANNOT ABORT after COMMIT

26

Summary: 2-phase locking variations

- 2PH: guarantees conflict serializable
- Strict 2PH: guarantees no cascaded aborts
- Conservative 2PH: guarantees no deadlock
- Strict + conservative 2PH: only allows reads of shared objects by uncommitted transactions.

27

How abort?

- Common: assume Strict 2PL

  => no cascaded aborts
- Keep log of all actions of all transactions:
  - Sequential writes on separate disk
  - Often write differences only
- To abort: undo actions of transaction backward in time using log
- Part algorithm for crash recovery

28

Crash Recovery Overview

- Goals of crash recovery
  - Either transaction commits and is correct or aborts
  - Commit means all actions of transaction have been executed
- Error model:
  - lose contents main memory
  - disk contents intact and correct

29

Crash recovery requirements

- If transaction has committed then still have results (on disk)
- If transaction in process, either
  1. Transaction completely aborts

  OR

  2. Transaction can continue after restore as if no crash
- Get serializable schedule such that transactions that committed before crash still commit and in same order

=> NEED LOG

30

Other consistency issues

Dynamics of DB can cause consistency problems even with Strict 2PL

Example: T1

1. lock all pages containing records with property P
2. Take an aggregate of those records
3. Lock all pages containing records with property Q
4. Take an aggregate of those records

T2

1. Lock new page
2. Insert new record with property P on new page
3. Lock new page
4. Insert new record with property Q on new page

Schedule: T1:1 T1:2 T2:1,2,3,4 T1:3 T1:4

Aggregate for P before T2 inserts; aggregate for Q after T2 inserts => not serializable and not consistent

31

Solutions?

- Need to lock all now and future records
- How?
  - Lock whole file : pages and access - **COSTLY**
  - Predicate locking: lock all records satisfying predicate (e.g. salary > 100K)
  - How?
    - Special case: if only using index to reach records satisfying predicate
    - Lock pages in index which contain or ***would contain*** data entries to records satisfying predicate

32