Making Snapshot Isolation Serializable

- Alan Fekete, University of Sydney
  - fekete@cs.usyd.edu.au
  - visiting MIT

- This is joint work with Patrick O'Neil, Elizabeth O'Neil and Dimitrios Liarokapis (all from UMass Boston) and Dennis Shasha (NYU)

Overview of Talk

- Concurrency Control background
- Snapshot Isolation background
- New results
  - when a mix of applications runs correctly with SI
  - if it does not
    * how to modify the applications to avoid the anomalies

OLTP Environment

- Data stored in DBMS
  - integrity constraints, not necessarily known to DBMS
- Static collection of application programs
  - possibly parameterized
  - can query and/or modify database
  - written to assume integrity conditions
    - and preserve them
- Users run application programs
  - same program can be run many times
    - perhaps with different parameters
    - perhaps concurrently

Concurrency Problems

- Interleaving of reads/writes by different applications can leave data not matching reality
  - even describing impossible reality
    - violation of integrity constraint
  - or lead to applications seeing inappropriate data
- Famous situations
  - lost update
    - transaction effects not present in final state
  - inconsistent read
    - application sees partial effects of a transaction

An Example

- domain: stock management
  - item X represents quantity in Sydney
  - item Y represents quantity in Perth
  - X+Y>100 is constraint system should keep
- application programs
  - T1 is transfer of 15 units from Sydney to Perth
    * read X into local x
    * read Y into local y
    * if x<15, then abort
    * subtract 15 from local x, add 15 to local y
    * write local x to X
    * write local y to Y

- More application programs
  - T2 is status check at Sydney
    - read X and display to user
  - T3 is dispatch of 10 units from Sydney
    - read X into local x
    - read Y into local y
    - if 100<=x+y<=110 then abort
    - subtract 10 from local x
    - write local x to X
  - T4 is dispatch of 10 units from Perth

Lost update

- Interleave T1 and T3, no concurrency control, start with X=50, Y=100
  - T1:read X into local x
  - T1:read Y into local y
  - T3:read X into local x
  - T3:read Y into local y
  - T3:check 100<=x+y<=110? No
  - T3:subtract 10 from local x
  - T3:write local x to X
  - T1:check x<15? No
  - T1:subtract 15 from local x, add 15 to local y
  - T1:write local x to X
  - T1:write local y to Y

Inconsistent Read

- Interleave T1 and T3, no concurrency control, start with X=25, Y=80
  - T1:read X into local x
  - T1:read Y into local y
  - T1:check x<15? No
  - T1:subtract 15 from local x, add 15 to local y
  - T1:write local x to X
  - T3:read X into local x
  - T3:read Y into local y
  - T3:check 100<=x+y<=110? No
  - T3:subtract 10 from local x
  - T3:write local x to X
  - T1:write local y to Y

Serializability Theory

- A history is *serializable*
  - provided it is equivalent (in impact on final state, and in outputs) to
  - an execution with each application running alone
    - one after another
- A key result: no matter what integrity constraint is considered,
  - if each application program acting alone maintains this constraint
  - then the constraint is true in the final state after a serializable history
- A nice theory to prove serializability by absence of cycles in dependency (conflict) graph

Drawbacks of Serializability

- The best known way to ensure serializable execution is strict two-phase locking
  - get locks before reading or writing
  - write lock on any item is exclusive
    - it prevents any other lock on that item
  - hold locks until application commits
- Locking can be efficiently implemented, but
  - application programs are often blocked
    - even read-only (common, should be fast)
  - so throughput is drastically lowered
- In practice, many sites run without holding readlocks
  - does not guarantee consistency to data
    - unless application programs are specially written

Snapshot Isolation

- A radical new concurrency algorithm
  - uses recovery log to allow time-travel
  - implemented in Oracle 7.3 (and later) as "isolation level serializable"
  - but does not actual provide serializability in all cases
- Read does not give current value
  - instead gives value as it was when transaction started
- First committer wins
  - check at transaction end that no concurrent transaction has committed modification to same item
    * if it did, abort instead
  - like optimistic concurrency control
    * but skips checks of items read and not written

Benefits of SI

- No extra storage for multiple versions
  - they are in the recovery log anyway
- Reading is never blocked, even by concurrent writer
  - throughput is good
- Prevents many bad situations
  - no “lost update”
    * because first-committer-wins
  - no “inconsistent read”
    * because all a transaction’s reads see the state at the time the transaction started

SI allows Skew Writes

- Consider the interleaved history of T3 and T4
  - start with X=50, Y=65
  - T3:read X into local x
  - T3:read Y into local y
  - T4:read X into local x
  - T4:read Y into local y
  - T3:check 100<=x+y<=110? No
  - T4:check 100<=x+y<=110? No
  - T3:subtract 10 from local x
  - T3:write local x to X
  - T4:subtract 10 from local y
  - T4:write local y to Y

Conflicts under SI

- "Read Dependency" (WR) U1 → U2
  - U1 modifies a data item that is seen by U2's read
    * or U1 modifies an item and U2's query reflects this in its matching set
  - U1 must completely precede U2 (because snapshot reads)
- "Write Dependency" (WW) U1 → U2
  - U1 modifies a data item that is later modified by U2
  - U1 must completely precede U2 (first-committer-wins)
- "Antidependency" (RW) U1 → U2
  - U1 reads a data item and does not see U2's modification
    * or U1 does a query and the matching set does not reflect U2's modification
  - either U1 completely precedes U2, or U1 and U2 are concurrent

Sometimes SI is OK

- If we run T1 and T3 with Oracle, no violation will happen
  - why? first committer wins means that they can't overlap
- In general, we can sometimes be sure from knowing the application programs that nothing will damage any integrity constraints
- Not only toy applications have this property
  - eg TPC-C benchmark
  - a complex mix of 5 programs dealing with a warehouse
- Our goal: the DBA can examine the application mix, and check whether it will necessarily give serializable executions
  - and therefore preserve integrity constraints

Static Analysis of Application Programs

- Examine the texts of application programs, to identify interactions between them
  - eg see which database tables are accessed or modified
- Based on knowing the fixed set of applications programs
  - doesn't work with dynamic SQL
  - doesn't work with ad-hoc access
- Based on interactions, our theory may allow DBA to be sure that every possible execution is serializable
  - if not, our theory pinpoints how to change the application programs to achieve guaranteed serializability

Conflicts

- Program P has a write-write conflict with Q ("(wP,wQ)")
  - in some history, there is a write dependency T $\rightarrow$ U
    * where T is execution of P; U is execution of Q
  - check if there is some item that P and Q might both modify
- Program P has a read-write conflict with Q "(rP,wQ)"
  - in some history, there is an antidependency T $\rightarrow$ U or a read-dependency U $\rightarrow$ T
    * where T is execution of P; U is execution of Q
  - check if there is some data item that Q can modify and P examines
    * also consider predicate conflict: Q inserts or deletes or modifies an item to change the set which matches a query in P
  - say the conflict is *vulnerable* if T and U can be concurrent

Conservative Analysis

- DBA must be conservative in analysis
  - include conflicts unless DBA is sure no execution could occur with common item
    * perhaps because where clauses are inconsistent
    * perhaps because control flow is incompatible
  - treat conflict as vulnerable unless DBA is sure concurrent execution can't give dependency/antidependency
    * perhaps due to workflow
    * perhaps due to first-committer-wins property of SI
- Ultra-conservative analysis can be automated
  - eg based only on the columns mentioned in SQL statements

Example analysis

- (wT1,wT4)
  - both may modify Y
- (rT2,wT1) is vulnerable
  - T2 reads X which T1 modifies
- (rT3,wT4) is vulnerable
  - T3 reads Y which T4 can modify
- (rT3,wT1)
  - T3 reads X and Y which T1 modifies
  - this is not vulnerable
    - as both modify X (first-committer-wins)

Produce a graph

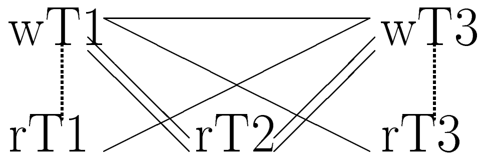
- Each program P gives two nodes: rP and wP
  - if P is read-only, have only rP
- join rP and wP by sibling edge (shown as dots)
- if P has write-write conflict with Q
  - join wP and wQ by conflict edge (shown solid)
- if P has vulnerable read-write conflict with Q
  - join rP and wQ by marked conflict edge (shown double)
- if P has not vulnerable read-write conflict with Q
  - join rP and wQ by conflict edge (shown solid)

Main Theorem

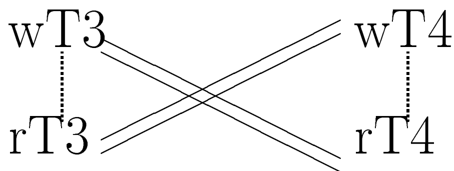
- Look for *dangerous* cycle in graph
  - cycle containing successive segments
    * vulnerable conflict then sibling then vulnerable conflict
      rP === wQ · · · rQ === wR
- If no dangerous cycle, then every SI-execution is serializable
- So any execution on DBMS with SI will preserve integrity constraints
- Corollary: if every vulnerable conflict involves a read-only transaction, then every SI-execution is serializable

Examples

- For the stock management example, consider T1, T2 and T3 (without T4)
  - no dangerous cycle



- For the stock management example, consider T3 and T4
  - there is a dangerous cycle



Proof Sketch I

- Lemma: A non-serializable execution has dependency graph with a cycle in which two consecutive edges are both between concurrent transactions
- Non-serializable execution implies cycle in dependency graph
- For non-serializable SI-execution, consider transaction in cycle with earliest commit time, and its two immediate predecessors in cycle: U1 $\rightarrow$ U2 $\rightarrow$ U3
  - U2 does not completely precede U3 (as U3 has earliest commit time)
    - so U2 and U3 are concurrent, with U2 start before U3 commits (this is antidependency)
  - U1 does not completely precede U2
    - f so, U1 would commit before U2 starts which is before U3 commits
    - so U1 and U2 are concurrent (this is antidependency)

Proof Sketch II

- Lemma: can lift cycle in dependency graph to cycle in program conflict graph
- Any edge in dependency graph between transactions corresponds to conflict between programs
  - concurrent transactions implies vulnerable conflict
- Fill in sibling edges as needed in lifting
- A complication: if a program may be instantiated multiple times
  - then twin each node in the graph: rP and rP' and wP and wP'
    - two copies of each is enough
- Proof of theorem: Non-serializable execution implies dependency graph cycle with two consecutive edges which involve concurrent transactions
  - the lifting of this must be dangerous cycle in program conflict graph

Avoiding Non-serializable Execution

- If the theorem doesn't show that the application will run correctly, what can the DBA do?
  - alter the application so the theorem does apply
    - without altering program semantics
- Change vulnerable edge in dangerous cycle, so it isn't vulnerable
  - this can make the cycle not dangerous
  - there are several ways to prevent vulnerable edges without changing application semantics

Preventing vulnerable conflicts

- DBA can introduce data item which is modified in both programs
  - to materialize the conflict, so first-committer-wins prevents concurrent execution
- Or, DBA can *promote* one data item which is read by P and written by Q, so P does identity write
  - first-committer-wins will prevent concurrent execution
  - in Oracle, “select for update” has same effect
- Try to promote as little as possible
  - if every read in every program is promoted, you get same commit-time test as optimistic concurrency control