Introduction to Database Systems

CSE 444

Lecture #10
Feb 7 2001

Announcements

- **Course Project MileStone2 due today**
- **Change in Deadlines**
    - **Homework#3 due on Feb 21**
    - **Project Report now due on Feb 28**
- **HW#2 has been linked**
    - **Constraints, Triggers, Security, Transactions**
- **MidTerm grading in progress**
    - **Feedback?...**

Concurrency Control

Reading: Sec 7.2, 9.1-9.3,9.4.1, 9.4.2,9.5, 9.6.3,10.3.1,10.3.2

Why Have Concurrent Processes?

- Better throughput, response time
- Done via better utilization of resources:
    - While one process is doing a disk read, another can be using the CPU or reading another disk.
- DANGER DANGER! Concurrency could lead to incorrectness!
    - Must carefully manage concurrent data access.
    - There's (much!) more here than the usual OS tricks!

Transactions

- Basic concurrency/recovery concept: a <u>transaction</u> (Xact).
    - A sequence of many actions which are considered to be one atomic unit of work.
- DBMS "actions":
    - (disk) reads, (disk) writes

The ACID Properties

- A tomicity: All actions in the Xact happen, or none happen
    - Account Transfer, Withdraw cash from ATM
- C onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent
- I solation: Execution of one Xact is isolated from that of other Xacts
    - Account Withdrawal
- D urability: If a Xact commits, its effects persist
    - Electronic Fund Transfer

Passing the ACID Test

- Concurrency Control
  - Guarantees Isolation
- Logging and Recovery
  - Guarantees Atomicity and Durability.
- We'll do C. C. first:
  - What is acceptable behavior?
  - What problems could arise?
  - How do we guarantee acceptable behavior?

Notation

- T1: Read(A,t), t:=t+100, Write(A,t), Read(B,t), t:= t + 300, Write(B,t)
- T2: Read(A,s), s:=s*2, Write(A,s), Read(B,s), S:=s*2, Write(B,s)
- T1: R1(A), W1(A), R1(B), W1(B)
- T2: R2(A), W2(A), R2(B), W2(B)
- What kind of interleaving makes sense?

Schedules

T1	***T2***
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

- Schedule: An interleaving of actions from a set of Xacts, where the actions of any 1 Xact are in the original order.
  - Represents some actual sequence of database actions.
  - Example: $R_1(A)$, $W_1(A)$, $R_2(B)$, $W_2(B)$, $R_1(C)$, $W_1(C)$
  - In a *complete* schedule, each Xact ends in commit or abort.
- Initial State + Schedule → Final State

Acceptable Schedules

- One sensible "isolated, consistent" schedule:
  - Run Xacts one at a time, in a series.
  - This is called a serial schedule.
  - NOTE: Different serial schedules can have different final states; all are "OK" -- DBMS makes no guarantees about the order in which concurrently submitted Xacts are executed.
- Serializable schedules:
  - Final state is what *some* serial schedule would have produced.
  - Aborted Xacts are not part of schedule; ignore them for now (they are made to `disappear' by using logging).

Serializability Violations

transfer $100 from A to B	add 6% interest to A & B
T1	***T2***
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

- Two actions may conflict when 2 xacts access the same item:
- Dirty Read (WR Conflict)
  - Result is not equal to any serial execution!
  - T2 reads what T1 wrote, but it shouldn't have!!
  - T1 still active!

Database is inconsistent!

Example: Dirty Read

- T1: Transfer $100 from A to B
- T2: Increment A and B by 6%
- Consider schedule
- R1(A) W1(A) R2(A) R2(B) W2(A) W2(B) R1(B) W1(B)

Serializability Violations (Contd.)

- Unrepeatable Read (RW Conflict)

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

- Lost Update (WW Conflict)

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

Examples: Unrepeatable Read/Lost Update

- Unrepeatable Read
  - T1: Increment A; T2: Decrement A
  - R1(A) R2(A) W1(A) W2(A)
- Lost Update/Blind Write
  - T1: Set salary of A,B to $10000
  - T2: Set salary of A,B to $30000
  - W1(A) W2(A) W2(B) W1(B)

Checking for Serializability

- Conflict: A pair of consecutive actions in a schedule such that
  - If their order is changed, then at least one of the transactions may change
- Non Conflicting Swaps
  - Unless actions within the same transaction
  - Unless actions on the same object
  - Unless one of the actions is a Write
    - WW: Wi(X), Wj(X)
    - RW: Ri(X), Wj(X)

Conflict Serializability

- Guarantees serializability
- 2 schedules are conflict equivalent if:
  - they have the same lists of actions, and
  - each pair of conflicting actions is ordered in the same way.
- A schedule is conflict serializable if it is conflict equivalent to a serial schedule.
  - Note: Some serializable schedules are not conflict serializable!

Example

- Example 9.6 from Text
- R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)

Example

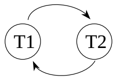
- All serializable schedules do not need to be conflict serializable
- Page 478 of Text
- S1: W1(Y), W1(X), W2(Y), W2(X), W3(X)
- S2: W1(Y), W2(Y), W2(X), W1(X), W3(X)

Test for Conflict Serializability: Precedence Graph

- A Precedence (or Serializability) graph:
  - Node for each committed Xact.
  - Arc from Ti to Tj if there is an action of Ti precedes and "conflicts" with an action of Tj
    - Ai before Aj
    - Ai and Aj involve the same database element
    - Either Ai or Aj is a WRITE
- Theorem 1: A schedule is conflict serializable iff its precedence graph is acyclic.

Example: Precedence Graph

- T1 transfers $100 from A to B, T2 adds 6%
  - R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B)



Example: Precedence Graph

- R1(A), W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)
- Is it conflict serializable?

Isolation Level

- Captures visibility across transactions
- Correct/Strongest Isolation Level
  - Serializable
  - Implemented through conflict serializability
    - Tested using precedence graph
- Weaker Isolation level
  - Dirty Read (RW)
  - Unrepeatable Read (WR)
- Choice of isolation level exposed through SQL
  - Discussed later in the lecture

Implementation of Serializability

Locking

- Concurrency control usually done via locking.
- Lock info maintained by a "lock manager":
  - Stores (XID, RID, Mode) triples.
    - Mode ∈ {S,X}
    - S for readers; X for writers
- Steps
  - Acquire Lock
    - If a Xact can't get a lock, it is suspended on a wait queue
  - Release Lock
- This is a simplistic view

Granting Lock Requests: Lock Compatibility

LOCK REQUESTED

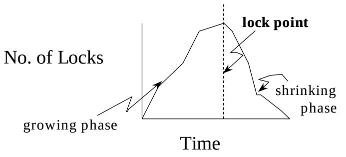
LOCK HELD

	--	S	X
--	√	√	√
S	√	√	
X	√		

Two-Phase Locking (2PL)

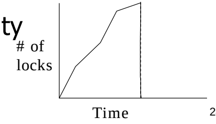
- 2PL:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - If T releases any lock, it can acquire no new locks!
- Locks are automatically obtained by DBMS.
- Guarantees serializability

Growing and Shrinking Phases of 2PL



Strict 2PL

- Strict 2PL:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - Hold all locks until end of transaction.
- Guarantees serializability



Conflict Serializability & 2PL

- Theorem 2: 2PL ensures that the precedence graph of the schedule will be acyclic
  - Guarantees conflict serializability (and serializability)
- Strict 2PL improves on this by ensuring recoverable schedules
  - More on Recovery in the next lecture

Example

- T1: R1(A), R1(B), W1(B)
- T2: R2(A), R2(B)
- Schedule:
- S1(A), R1(A), S2(A), R2(A), S2(B), R2(B), X1(B)-denied, U2(A), U2(B), X1(B), R1(B), W1(B), U1(A), U2(B)

Deadlocks

- Deadlock: A set of lock requests waiting for each other
- System intervention necessary
- 2PL cannot prevent deadlocks
- Break deadlock by aborting one of the transactions

Example

- Consider the sequence of actions:
  - R1(X) R2(Y) W2(X) W1(Y)

Detecting Deadlock

- Timeout
- Graph-Based Detection (Chapter 10.3.1-.2)
  - Build a waits-for graph
    - Node = Transaction
    - Add Edge = Waiting situation; edge(T1,T2) if T1 is waiting on a lock held by T1
    - Delete Edge = Unblocking
    - Cycle = Deadlock
    - Check periodically for cycles
- Example: R1(X) R2(Y) W2(X) W1(Y)

The Phantom Problem

- T1 locks all pages containing sailor records with *rating* = 1, and finds <u>oldest</u> sailor (say, *age* = 71).
- T2 inserts a new sailor; *rating* = 1, *age* = 96.
- T2 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 <u>oldest</u> (say, *age* = 63)

Phantom Problem: Analysis

- The schedule is not serial but 2PL would allow such a schedule?
- 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!
  - The sailor with rating 1, age 96 is a <u>*phantom tuple*</u>
- Observation
  - Ensure that the "right" objects are locked
  - E.g., use predicate locks
  - No change in 2PL needed

Implementing Locking

- Needs to execute Lock and Unlock as atomic operations
- Needs to be very fast ~100 instructions
- Lock Table
  - Low-level data structure in memory (not SQL Table!)
  - Implemented as a hash table

Issues in Managing Locks

- Multi-granularity locking
  - Concurrency v.s. locking overhead
  - Intention locks on higher-level objects
  - Lock Escalation
- Hot spots
  - Minimize lock duration

SQL-92 Syntax for Transactions

- Start Transaction: No explicit statement. Implicitly started
  - By a SQL statement
  - TP monitor (agents other than application programs)
- End Transaction:
  - By COMMIT or ROLLBACK
  - By external agents

SQL-92: Setting the Properties of Transactions

- SET TRANSACTION
  - [READ ONLY | READ WRITE]
  - ISOLATION LEVEL

    [READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ | READ COMMITTED]
  - DIAGNOSTICS SIZE

    Value_Specification

Explanation of Isolation Levels

- Read Uncommitted
  - Can see uncommitted changes of other transactions
  - Dirty Read, Unrepeatable Read
  - Recommended only for statistical functions
- Read Committed
  - Can see committed changes of other transactions
  - No Dirty read, but unrepeatable read possible
  - Acceptable for query/decision-support
- Repeatable Read
  - No dirty or unrepeatable read
  - May exhibit *phantom* phenomenon
- Serializable

Implementation of Isolation Levels

ISOLATION LEVEL	DIRTY READ	UNREPEATABLE READ	PHANTOM	IMPLEMENTATION
Read Uncommitted	Y	Y	Y	No S locks; writers must run at higher levels
Read Committed	N	Y	Y	Strict 2PL X locks; S locks released anytime
Repeatable Reads	N	N	Y	Strict 2PL on data
Serializable	N	N	N	Strict 2PL on data and indices (or predicate locking)

Summary of Concurrency Control

- Concurrency control key to a DBMS.
- Transactions and the ACID properties:
  - I handled by concurrency control.
  - A & D coming soon with logging & recovery.
- Conflicts arise when two Xacts access the same object, and one of the Xacts is modifying it.
- Serial execution is our model of correctness.

Summary of Concurrency Control (Contd.)

- Serializability allows us to “simulate” serial execution with better performance.
- 2PL: A simple mechanism to get serializability.
- Lock manager module automates 2PL
    - Lock table is a big main-mem hash table
- Deadlocks are possible, and typically a deadlock detector is used to solve the problem.