

Overview of Transaction Management 2

19 January 2010
Lecture 14

Our Topics Today

- Two important tasks the database does:
 - **Concurrency Control**: Let multiple users use the database at once
 - **Crash Control**: Let transactions/database fail gracefully
- Failures can be:
 - Transactions that don't complete
 - Loss of power
 - File corruption
 - Disk failure
- Fundamental concept: A **transaction**

January 19, 2010

ISE 322: Database Systems

2

Outline

- Concurrent Execution of Transactions
- Lock-based concurrency control
- Performance of Locking
- Source: R&G Chapter 16.1-16.5

January 19, 2010

ISE 322: Database Systems

3

What is a transaction?

- A **transaction** is **one execution of a user program on a DBMS**
 - Multiple program executions → multiple transactions
 - Multiple users → multiple transactions
 - To the database, a transaction is a series of reads/writes
- If it finishes: **COMMIT**. Otherwise it fails: **ABORT**
- Almost all DBMS' **interleave** multiple transactions
 - So they must provide **guarantees** about behavior (concurrency)
 - They must protect the transactions from each other (concurrency)
 - If one transaction fails, the others must be protected (crash)
 - **Why interleave?**
 - To improve database transaction throughput
 - Use latencies that arise during execution of individual transactions

January 19, 2010

ISE 322: Database Systems

4

ACID

- Four properties:
 - **Atomicity**
 - **Consistency**
 - **Isolation**
 - **Durability**
- Define how DBMS' offer **concurrency** and **crash** control
- The **log** is essential here (we'll talk more about it later)

January 19, 2010

ISE 322: Database Systems

5

Serializability

- Essential concept: **Serializability**
 - Schedule S1 is **serializable** if we can find some (different) schedule on the transactions S2 which is **serial** and is the **same** as S1
 - Serializability is a property of **schedules**

What does "the same" mean?

- For now – identical outcomes (**view serializability**):
 - The order of actions in each transaction is identical
 - If T reads value $X=x_i$ in S1, it reads $X=x_i$ in S2 (and not some other value $X=x_j$)
 - If at the end of S1, $X = x_i$, then $X = x_i$ at the end of S2

January 19, 2010

ISE 322: Database Systems

6

Example: Interleaving 1

Example: The schedule below is serializable:

T_1	T_2
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
$R(B)$	
$W(B)$	
	$R(B)$
	$W(B)$
Commit	Commit

It is equivalent to running $T_1; T_2$.

Example: Interleaving 2

Example: The following schedule is also serializable, but in a different way than the previous one:

T_1	T_2
	$R(A)$
	$W(A)$
$R(A)$	
	$R(B)$
	$W(B)$
$W(A)$	
$R(B)$	
$W(B)$	
Commit	Commit

It is equivalent to running $T_2; T_1$.

Enforcing Serializability

A DBMS *may* execute a transaction schedule even if they are **not serializable**:

- It may be equivalent to *another* serializable schedule
- The DBA or transaction can tell the DBMS to not care about serializability

Defining Serializability: Anomalies

Serializability is more complicated than ordering – we need to examine **data flows**

- Some data flows generate *anomalies*

Anomalies are situations which may break ACID

- If they do, they are called *conflicts*
- They are common to all transaction management applications

Conflicts reflect unintended **overlap** or interaction of transactions

We describe anomalies in terms of T_1 and T_2 based on their order of reading and writing

1. Write-Read (WR) – T_2 read something T_1 wrote
2. Read-Write (RW) – T_2 wrote something on top of what T_1 read
3. Write-Write (WW) – T_2 wrote something on top of what T_1 wrote

Reading Uncommitted Data (WR)

- T_2 reads data written by T_1 but uncommitted

- Data read is *dirty* since T_1 has written but not committed

- What's the problem?

- Remember the contract for Consistency

- Transactions only required to guarantee consistent state **when they commit**

- T_2 may read inconsistent data since T_1 didn't finish yet

T_1	T_2
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

Unrepeatable Reads (RW)

- When T_1 reads A , but a later read to A yields a **different** value, since T_2 has written A

- By **Isolation**, transactions assume that any value read will remain unchanged

Example: A is the number of copies of a book in stock.

T_1 and T_2 both want to order the book.

T_1 read $A=1$, so it wants to order.

T_2 also read $A=1$ and orders the book ($A=0$).

T_1 then tries order ($A = -1$)

T_1	T_2
$R(A)$	
	$R(A)$
	$W(A)$
	Commit
$W(A)$	
Error	

Overwriting Uncommitted Data (WW)

When T_2 overwrites data from another transaction T_1 that it is interleaved with, the updates from T_1 are **lost**

- May break **Consistency**
- Called **lost update**

Example: A is a number which must be kept as twice B.

T_1 and T_2 update A and B.

T_1 will set A=10 and B=5.

T_2 will set A=12 and B=6.

At the end, the values will be A=12, B=5 since T_2 overwrote T_1 's value for A.

T_1	T_2
W(A)	W(A)
	W(B)
W(B)	Commit
Commit	

January 19, 2010

ISE 322: Database Systems

13

What about Aborts?

- Transactions which abort should have their effects **undone**
 - No committed transactions should reflect data from an aborted one
- Not always possible to totally undo effects
 - Called "**recoverability**" which we'll talk about later
 - Simply undoing actions can be dangerous if they were later rewritten by other transactions
- **Extended definition of serializability:** A serializable S1 schedule is one which is the same as some serial execution S2 of *the transactions which have committed*.
 - Meaning S2 shouldn't include the aborted transactions from S1

January 19, 2010

ISE 322: Database Systems

14

Cascading Aborts

If T_2 read data from T_1 and then T_1 aborted, we have two choices

- Let T_2 continue (and possibly commit)
- Forcibly abort T_2

Cascading aborts: Aborting one transaction because it was affected by or read data from another one that aborted

Cascading aborts are **bad** – abort T_2 because T_1 aborted?

- Solution: **Enforce that transactions only read from committed transactions**
- A schedule which preserves this property **avoids cascading aborts**

January 19, 2010

ISE 322: Database Systems

15

Recoverability

What if T_2 read from T_1 , T_2 committed, and then T_1 aborts?

- Durability! Consistency!

- We are stuck!

Solution: Don't let a transaction commit unless all of the transactions it read from already have

Recoverable schedule: ensures that transactions commit only after the transactions they have read from have committed

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Abort	Commit

January 19, 2010

ISE 322: Database Systems

16

So far

- Concurrent Execution of Transactions
- Lock-based concurrency control
- Performance of Locking

January 19, 2010

ISE 322: Database Systems

17

Enforcing ACID

- How do we apply the theories?
- Can we enforce Serializability? Recoverability?
- The most common tool: **Locks**
 - A **lock** is a small bookkeeping object which is associated with the object
- We use locks as part of a **Locking Protocol**
 - A **Locking protocol** is a set of rules followed for managing the locks
- This semester we'll talk about: **Strict Two Phase Locking**
 - **More next semester**

January 19, 2010

ISE 322: Database Systems

18

Strict Two-Phase Locking

- Abbreviated Strict 2PL
- Kinds of Locks:
 - Exclusive Lock - allows reading and writing
 - Shared Lock – allows just reading
- Three rules in Strict 2PL
 1. If transaction wants to **read** an object, it must first get a **shared** lock
 2. If transaction wants to **write** an object, it must first get an **exclusive** lock
 3. All locks held by a transaction are released when the transaction is **completed** (commit or abort)
- The DBMS manages the lock requests and releases

January 19, 2010

ISE 322: Database Systems

19

Properties of Strict 2PL

- Strict 2PL is **conservative** and **pessimistic**
 - It is not possible to break Serializability or ACID while following it
 - Conversely, many serializable schedules are forbidden by it
- Strict 2PL prevents all (true) transaction conflicts
 - If two transactions do not conflict, they can execute with interleaving
 - If not (i.e. they overlap), they will need to execute serially
- More notation:
 - Request action for a shared lock on O by T as $S_T(O)$
 - Request action for an exclusive lock on O by T as $X_T(O)$

January 19, 2010

ISE 322: Database Systems

20

Example 1: Strict 2PL

T_1	T_2	T_1	T_2
$R(A)$		$S(A)$	
	$R(A)$	$R(A)$	
	$R(B)$		$S(A)$
	$W(B)$		$R(A)$
	Commit		$X(B)$
$R(C)$			$R(B)$
$W(C)$		$X(C)$	$W(B)$
Commit		$R(C)$	Commit
		$W(C)$	
		Commit	

January 19, 2010

ISE 322: Database Systems

21

Example 2: Strict 2PL

T_1	T_2	T_1	T_2
		$X(A)$	
$R(A)$		$R(A)$	
$W(A)$		$W(A)$	
	$R(A)$	$X(B)$	
	$W(A)$	$R(B)$	
	$R(B)$	$W(B)$	
	$W(B)$	Commit	
	Commit		$X(A)$
$R(B)$			$R(A)$
$W(B)$			$W(A)$
Commit			$X(B)$
			$R(B)$
			$W(B)$
			Commit

January 19, 2010

ISE 322: Database Systems

22

Deadlocks

- **Deadlocks** are a problem in most locking frameworks
 - Occur when two transactions are trying to get locks held by each other

T_1	T_2
$X(A)$	
	$X(B)$
$X(B)$	
	$X(A)$

- **Solution:** Abort one of them
- **Prevention:**
 - Fixed lock acquisition orders (must be done by the program)
 - Minimize the length of transactions (so they hold locks for less time)

January 19, 2010

ISE 322: Database Systems

23

Performance Price for Locks

Locks **require processing time**

- They ensure serializability and ACID by either **blocking or aborting** transactions
- Deadlocks tend to be **rare** (in well designed programs), so we don't worry too much about them
- The more transactions there are, the more time each of them may spend **waiting for locks**
 - At some point allowing more transactions to start will make them all go slower – called **Thrashing**
- Techniques to **increase throughput**:
 - Reduce the **object size** (to prevent false conflicts)
 - Reduce the **time** transactions hold locks (e.g. keep them short)
 - Reduce objects which are **frequently accessed** by lots of transactions

January 19, 2010

ISE 322: Database Systems

24

In summary

- Concurrent Execution of Transactions
- Lock-based concurrency control
- Performance of Locking