

---

---

# Review of Transaction Management, Transactions and SQL

21 February 2011  
Lecture 1

---

# 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**

# Outline

---

- ACID Properties
  - Transactions and Schedules
  - Concurrent Execution of Transactions
  - Lock-based concurrency control
  - Performance of Locking
  - Transaction Support in SQL
- 
- Source: R&G Chapter 16.1-16.6

# 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

# 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)



*“As always, should you or any of your IM force be caught or killed, the Secretary will disavow any knowledge of your actions.”*

*“Good luck, Jim. This tape will self-destruct in five seconds.”*

# Atomicity

---

- Transactions are either executed entirely or not at all
- Some transaction cannot complete due to:
  - Internal failure, Power failure
  - Crash, Unexpected property discovered

Such transactions **abort**

The database's job: **ensure aborted transactions “never occurred”**

- Means undoing everything they did (use the log)
- Nobody else should have used data written by it (?)

Otherwise, an aborted transaction may leave the database in an incorrect state

# Consistency

---

*“Be kind, please rewind”* – Blockbuster video

**Consistency** is a **high level database property**

- Data has some “correct” or “logical” state which ought to be preserved
  - Ex. All employees are in the Employees table
  - Ex. The salaries in the Employees table are correct

The database **can't** enforce them – **it's the user's job**

- Consistency is a **contract**:
  - The database promises that transactions will **start with a consistent database**
  - Each transaction must promise that **if it receives a consistent database, then it will return a consistent database at completion**
    - What if the transaction **doesn't complete**?
    - **How** can transactions promise this?

# Isolation

---

*“Have you ever been alone in a crowded room”* – Jack’s Mannequin

**Isolation** means that even though many transactions may be going on at once:

- The transactions **can’t tell** (think time sharing systems)
  - Result of the many concurrent transactions is **equivalent to some serial execution** of them all (ideally)
- **Example:** T1 and T2 execute at the same time.
    - Both complete **successfully**
    - T1 **couldn’t tell that T2 ran at the same time** or vice versa
    - After they complete, to an outsider **it looks like one ran after the other** (no guarantee if it’s T1;T2 or T2;T1)

# Durability

---

**Durability** means that if a transaction commits its effects are **never lost**



Conversely, an aborted transaction is eventually forgotten

- The log is used for this
  - Must be safe from crashes
  - Must reflect every committed transaction (at least)

# ACID Together

---

Putting ACID together, we get:

- Transactions are atomic collections of reads/writes
- They run in simulated isolation
- If one finishes (commits), its changes are never lost
- If one doesn't finish (aborts), an outsider couldn't tell it ever ran
- To an outsider, it appears that all successful transactions could have run serially
- Transactions always receive a consistent database (as per the contract)

# So far

---

- ACID Properties
- Transactions and Schedules
- Concurrent Execution of Transactions
- Lock-based concurrency control
- Performance of Locking
- Transaction Support in SQL

# Schedules and Notation

---

- We may show the running of several transactions in a “**schedule**”
  - Time is shown from top to bottom
  - Each transaction is shown in a separate column
  - A **complete schedule** shows commits/aborts for all transactions
  - A **serial schedule** has no interleaving
- If T reads O we write  $R_T(O)$
- If T writes O we write  $W_T(O)$
- A completed transaction sends **commit**:  $Commit_T$
- If not **abort**:  $Abort_T$
- We will drop the subscript if it's obvious

# Schedule Model

---

## Two assumptions

- Transactions communicate only via database
  - Important for isolation and durability (why?)
  - We can't help if two transactions communicate in an out of band manner (why would we?)
- A database is a fixed collection of independent object
  - Addition and deletion complicate things
    - We'll relax this later
  - Dependency also complicates (locks)
- What's an object?

# A Simple Schedule

---

---

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

# Serializability

---

- Essential concept: **Serializability**
    - Schedule  $S_1$  is *serializable* if we can find some (different) schedule on the transactions  $S_2$  which is **serial** and is the **same**  $S_1$
    - 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  $X=x_1$  in  $S_1$ , it reads  $X=x_1$  in  $S_2$  (and not some other value  $X=x_2$ )
  - If at the end of  $S_1$ ,  $X = x_1$ , then  $X = x_1$  at the end of  $S_2$

# 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**
  - T2 may read inconsistent data since T1 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)
	Commit
W(B)	
Commit	

# 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

# Cascading Aborts

---

If T2 read data from T1 and then T1 aborted, we have two choices

- Let T2 continue (and possibly commit)
- Forcibly abort T2

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

Cascading aborts are **bad** – abort T2 because T1 aborted?

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

# Recoverability

---

What if T2 read from T1, T2 committed, and then T1 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)
	Commit
Abort	

# So far

---

- ACID Properties
- Transactions and Schedules
- Concurrent Execution of Transactions
- Lock-based concurrency control
- Performance of Locking
- Transaction Support in SQL

# 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**

# 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

# 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  $X_T(O)$

# Example 1: Strict 2PL

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

  

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

# Example 2: Strict 2PL

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

# 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)

# 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

# So far

---

- ACID Properties
- Transactions and Schedules
- Concurrent Execution of Transactions
- Lock-based concurrency control
- Performance of Locking
- **Transaction Support in SQL**
- **Introduction to Crash Recovery**

# Transaction Support in SQL

---

- A Transaction **starts** when an application connects with DBMS and access database
    - Reads: SELECT, UPDATE, DELETE
    - Write: CREATE TABLE, INSERT, UPDATE, DELETE
  - A Transaction **ends** when:
    - The program sends “COMMIT” or “ROLLBACK” or
    - The program’s connection to the database server is dropped
- 
- If you don’t explicitly declare it – **each command (or group) sent is a separate transaction**
    - No need for commit/rollback
  - To explicitly declare a group of commands as a single transaction:
    - **BEGIN TRANSACTION**
    - **COMMIT/ROLLBACK**

# SQL:1999: Savepoints

---

- To better support long running transactions, use Savepoints
  - It is a bookmark which is declared
  - SAVEPOINT s1
- A transaction can partially rollback to a savepoint:
  - ROLLBACK TO SAVEPOINT s1
  - Will undo changes since Savepoint s1
  - Undoes declaration of s1
  - Automatically releases all locks acquired since
- Example of *nested transactions*

$T_1$

---

$X(A)$   
 $R(A)$   
 $W(B)$   
SAVEPOINT s1  
 $X(B)$   
 $R(B)$   
 $W(B)$   
SAVEPOINT s2  
 $S(C)$   
 $R(C)$   
 $W(A)$   
 $W(B)$   
ROLLBACK TO s1

# Savepoints in MS SQL Server

---

```
SAVE { TRAN | TRANSACTION } { savepoint_name |  
    @savepoint_variable }
```

- Can rollback to the savepoint using:  
    ROLLBACK TRANSACTION savepoint\_name
- Good for try/catch blocks
  - Try something which may fail and just undo it if it does
- Unlike SQL:1999, partial rollback **does not** release acquired locks

# Transaction Chaining

---

- In SQL:1999, a short cut to starting a new transaction when finishing one
  - Keeps the **same modes and settings** from the previous one
- “Chain” them:
  - COMMIT AND CHAIN
  - ROLLBACK AND CHAIN
- All locks are released from the previous transaction, but the settings are preserved

**Not supported in MS SQL. Supported in Oracle.**

# Locking Objects

---

- We talked about locking “objects”. What is an object?
- Example:

```
SELECT S.rating, MIN(S.age)
FROM Sailors S WHERE S.rating = 8
```
- What should the query lock?
  - Entire Sailors table?
  - Just the rows that have *rating* = 8?
- What if  $T_2$  wants to update age of a sailor called Joe who has rating =8?
- Shared-locked the whole table for the query until finished?
  - No other queries could update Sailors

# Table vs Row Locks

---

---

- One option is *Table Locks* (i.e. Lock all of Sailors)
- 
- Another option: shared lock the rows that  $T_1$  wants to read
    - That means lock all of the rating = 8 rows
    - Better performance
  - Row level locking is common
    - Called locking based on a *selection predicate*

# Another Example

---

- What about a transaction which examines the whole table and then updates just some of the rows?

```
SELECT S.rating as ratinggroup, S.name, MIN(S.age) as  
  minage  
FROM Sailors S  
GROUP BY S.rating, S.name  
HAVING COUNT(*) > 1
```

- We need to look over the entire table and do grouping
- May need to lock the whole table with shared lock and lock rows we update with exclusive locks

# Conclusion

---

- ACID Properties
- Transactions and Schedules
- Concurrent Execution of Transactions
- Lock-based concurrency control
- Performance of Locking
- Transaction Support in SQL