

TRANSACTION MANAGEMENT

CS 564- Fall 2016

ACKs: Jeff Naughton, Jignesh Patel, AnHai Doan

EXAMPLE

```
Read(A);  
Check (A > $50);  
Pay($25);  
A := A - 25;  
Write(A);
```

- Start with \$100
- What happens if the DBMS crashes right after we pay?
- What can happen if we interleave the execution of two such programs?

TRANSACTION MANAGEMENT

- **Inconsistencies** can occur when:
 - interleaving actions of different user programs
 - system crash, user abort
- Why not admit only one query into the system at any time?
 - lower utilization: CPU/IO overlap
 - long running queries starve other queries
- Provide the users with an illusion of a single-user system

TRANSACTION

- A collection of operations that form a single *atomic* logical unit

```
BEGIN TRANSACTION
    {SQL}
END TRANSACTION
```
- Operations:
 - READ(X), WRITE(X): X is a tuple
 - Special actions: COMMIT, ABORT
- Transactions must leave the database in a consistent state

THE ACID PROPERTIES

Atomicity: All actions in the transaction happen, or none happen

Begin

Read(A);

A := A - 25;

Write(A);

Read(B);

B := B + 25;

Write(B);

Commit

- **Example:** if the system crashes after Write(A), we **undo** the actions of the transactions

THE ACID PROPERTIES

Consistency: a database in a consistent state will remain in a consistent state after the transaction

Begin

Read(A);

A := A - 25;

Write(A);

Read(B);

B := B + 25;

Write(B);

Commit

- **Example:** $A+B$ must remain the same after the transaction is executed

THE ACID PROPERTIES

Isolation: the execution of one transaction is isolated from other (possibly interleaved) transactions

- if T1, T2 are interleaved, the result should be the same as executing first T1 then T2, or first T2 then T1

THE ACID PROPERTIES

Durability: if a transaction **commits**, its effects must persist

- for example, if the system crashes after a commit, the effects must remain
- what happens if the modified data is not written on disk?

SCHEDULES

- **Schedule:** An interleaving of actions from a set of transactions, where the actions of any one transaction are in the original order
 - **complete** schedule: each transaction ends in commit or abort
 - **serial** schedule: no interleaving of actions from different transactions

WHAT IS A GOOD SCHEDULE?

Serializable schedule:

- final state is what *some* **complete serial** schedule of committed transactions would have produced
- Can different serial schedules have different final states?
 - Yes, there is no specific ordering
- Aborted transactions?
 - ignore them for a little while (can be made to ‘disappear’ using logging)

SERIALIZABILITY VIOLATIONS

When execution of transactions is interleaved, we can have 3 different violations:

- Write-Read conflict (dirty read)
- Read-Write conflict (unrepeatable read)
- Write-Write conflict (overwriting uncommitted data)

DIRTY READ

@Start (A,B) = (1000, 100)

- Interleaved execution:
 - (990, 210)
- T1 → T2:
 - (900, 200) → (990, 220)
- T2 → T1:
 - (1100, 110) → (1000, 210)

<i>T1: Transfer \$100 from A to B</i>	<i>T2: Add 10% interest to A & B</i>
begin	
	begin
R(A) ; A -= 100	
W(A)	
	R(A) ; A *= 1.1
	W(A)
	R(B) ; B *= 1.1
	W(B)
	commit
R(B) ; B += 100	
W(B)	
commit	

UNREPEATABLE READ

- T1 reads value A: $R_{T1}(A)$
- T2 interleaves and overwrites the value: $W_{T2}(A)$
- T1 reads again: $R_{T1}(A)$ but sees a different value!

OVERWRITING UNCOMMITTED DATA

- T2 overwrites what T1 wrote!
- Example:
 - suppose that students in the same group must get the same project grade
 - T1: W (X=A), W (Y=A)
 - T2: W (X=B), W (Y=B)
 - $W_{T1}(X=A) \rightarrow W_{T2}(X=B) \rightarrow W_{T2}(Y=B) \rightarrow W_{T1}(Y=A)$

ABORTED TRANSACTIONS

- A **serializable** schedule is equivalent to a serial schedule of **committed** transactions
 - as if aborted transactions never happened!
- Two issues:
 - How does one undo the effects of a transaction?
 - by logging/recovery
 - What if another transaction sees these effects??
 - Must undo that transaction as well!

CASCADING ABORTS

- *cascading abort*: when abort of T1 requires an abort of T2
- What happens if T2 has already committed?
- *recoverable* schedule: Commit only after all transactions that supply dirty data have committed
- *ACA (avoids cascading abort)* schedule:
 - transaction only reads committed data
 - no cascading aborts can arise!

LOCKING

- Locking is a technique for **concurrency control**
- Lock information maintained by a *lock manager*:
 - stores (TID, RID, Mode) triples
 - Mode is either Shared (S) or Exclusive (X)

	--	S	X
--	✓	✓	✓
S	✓	✓	
X	✓		

- If a transaction cannot get a lock, it has to wait in a queue

STRICT 2 PHASE LOCKING

- Each transaction must obtain a **S** lock on object before reading, and an **X** lock on object before writing
- All locks held by a transaction are released only when the transaction completes
- If a transaction holds an **X** lock on an object, no other transaction can get a lock (S or X) on that object

Strict 2PL guarantees **serializability** and **ACA!**

NON-STRICT 2 PHASE LOCKING

- Each transaction must obtain a **S** lock on object before reading, and an **X** lock on object before writing
- If the transaction releases any lock, it can not acquire any additional locks

Non-Strict 2PL guarantees **serializability** (but not ACA)

EXAMPLE

Blackboard!

DEADLOCKS

- Example:

$X_{T_1}(B), X_{T_2}(A), S_{T_1}(A), S_{T_2}(B)$

- Deadlocks can cause the system to wait forever
- We need to detect deadlocks and break, or prevent deadlocks
- Simple mechanism: timeout and abort
- More sophisticated methods exist

PERFORMANCE OF LOCKING

- Locks have a performance penalty:
 - **blocked** actions
 - **aborted** transactions
- Because of blocking, we can not increase forever the throughput of transactions
- At the point where the throughput cannot increase, we say that the system **thrashes**

TRANSACTIONS IN SQL

- Transaction boundary
 - begins implicitly when a statement is executed
 - ends by COMMIT or ROLLBACK
- For long running transactions, we can use SAVEPOINT
 - we can then roll back to any previous savepoint

TRANSACTIONS IN SQL

- What object should we lock?

```
SELECT COUNT(*)  
FROM Employee  
WHERE age = 20 ;
```

- We can apply locking at different **granularities**:
 - lock the whole table Employee
 - lock only the rows with age = 20

THE PHANTOM PROBLEM

- So far we have assumed the database to be a static collection of elements (=tuples)
- If tuples are inserted/deleted then the **phantom problem** appears
- **Example:** blackboard!

TRANSACTIONS IN SQL

Transaction characteristics:

- Access mode: READ ONLY, READ WRITE
- Isolation level
 - **Serializable**: default (Strict 2PL)
 - **Repeatable reads**: (R/W locks, but phantom can occur)
 - Read only committed records
 - Between two reads by the same transaction, no updates by another transaction
 - **Read committed** (W locks longterm, R locks shortterm)
 - Read only committed records
 - **Read uncommitted** (only reads, no locks)

CRASH RECOVERY

Motivation:

- **Atomicity**: transactions may abort (rollback)
- **Durability**: the DBMS may crash

Buffer pool strategies:

- **Force**: every write goes to disk once committed
 - poor response time
 - provides durability
- **Steal**: buffer pool frames write to disk before commit

STEAL AND FORCE

STEAL (why enforcing Atomicity is hard)

- *To steal frame F*, current page in F (say P) is written to disk; some transaction holds lock on P
 - What if the transaction with the lock on P aborts?
 - We must remember the old value of P at steal time (to support **UNDO**ing the write to page P)

NO FORCE (why enforcing Durability is hard)

- what if we crash before a modified page is written to disk?
- write as little as possible, in a convenient place, at commit time, to support **REDO**ing modifications.

LOGGING

- Record **REDO** and **UNDO** information for every update in a **log**
- **Log**: An ordered list of REDO/UNDO actions
- The Write-Ahead Logging (WAL) protocol:
 - force the log record for an update before the corresponding data page gets to disk (guarantees **atomicity**)
 - write all log records for a transaction before commit (guarantees **durability**)

ARIES

- **ARIES** is a recovery algorithm that works with a steal, no-force approach
- Three phases:
 - Analysis
 - UNDO
 - REDO
- For more on crashes and recovery, take CS 764!

RECAP

- Transaction management
- **ACID** properties
 - atomicity
 - consistency
 - isolation
 - durability
- Techniques for transaction management
 - aborts, locking (2PL, strict 2PL)
- Crash Recovery