Lecture 6

Database Management Systems

Winter 2004

CMPUT 391: Properties of Transactions

Dr. Osmar R. Zaïane

University of Alberta

Chapter 20 of Textbook

Objectives of Lecture 6

Properties of Transactions

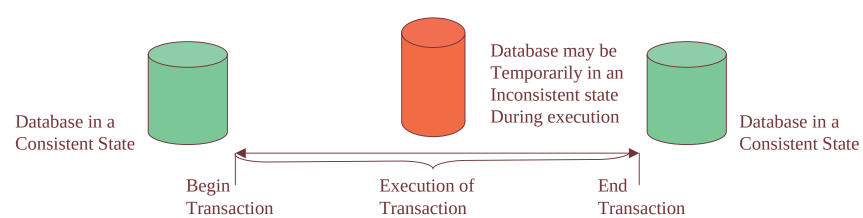
- Introduce some important notions related to DBMSs such as transactions, scheduling, locking mechanisms, committing and aborting transactions, etc.
- Understand the issues related to concurrent execution of transactions on a database.
- Present the properties of transactions

Transactions

- Many enterprises use databases to store information about their state
  - *e.g.*, Balances of all depositors at a bank
- When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way
  - *e.g.*, Bank balance must be updated when deposit is made
- Such a program is called a **transaction**

Transaction

- A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes
- A transaction is a sequence of actions that make consistent transformations of system states while preserving system consistency



What Does a Transaction Do?

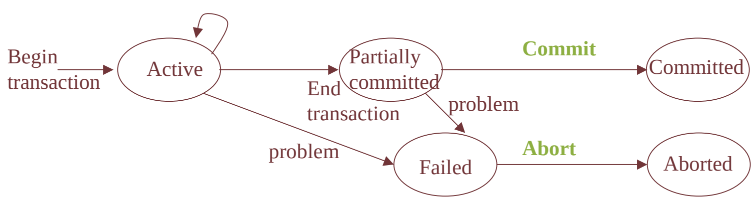
- Update the database to reflect the occurrence of a real world event
  - Deposit transaction: Update customer's balance in database
- Cause the occurrence of a real world event
  - Withdraw transaction: Dispense cash (and update customer's balance in database)
- Return information from the database
  - RequestBalance transaction: Outputs customer's balance

Transaction Operations

- A user's program may carry out many operations on the data retrieved from DB but DBMS is only concerned about Read/Write.
- A database transaction is the execution of a program that include database access operations:
  - Begin-transaction
  - Read
  - Write
  - End-transaction
  - Commit-transaction
  - Abort-transaction
  - Undo
  - Redo
- Concurrent execution of user programs is essential for good DBMS performance.

State of Transactions

- Active: the transaction is executing.
- Partially Committed: the transaction ends after execution of final statement.
- Committed: after successful completion checks.
- Failed: when the normal execution can no longer proceed.
- Aborted: after the transaction has been rolled back.



Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Transactions

- The execution of each transaction must maintain the relationship between the database state and the enterprise state
- Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

  } ACID properties

Transaction Properties

The acronym ACID is often used to refer to the four properties of DB transactions.

- **A**tomicity (all or nothing)
  - A transaction is *atomic*: transaction always executing all its actions in one step, or not executing any actions at all.
- **C**onsistency (no violation of integrity constraints)
  - A transaction must preserve the consistency of a database after execution. (responsibility of the user)
- **I**solation (concurrent changes invisible→serializable)
  - Transaction is protected from the effects of concurrently scheduling other transactions.
- **D**urability (committed updates persist)
  - The effect of a committed transaction should persist even after a crash.

ACID

Durability

- The system must ensure that once a transaction commits, its effect on the database state is not lost in spite of subsequent failures
  - Not true of ordinary programs. A media failure after a program successfully terminates could cause the file system to be restored to a state that preceded the program's execution

ACID

Implementing Durability

- Database stored redundantly on mass storage devices
- Architecture of mass storage devices affects type of media failures that can be tolerated
  - **Availability**: extent to which a (possibly distributed) system can provide service despite failure
    - Non-stop DBMS (mirrored disks)
    - Recovery based DBMS (log)

ACID

Isolation

- **Serial Execution**: The transactions execute one after the other
  - Each one starts after the previous one completes.
  - The execution of each transaction is **isolated** from all others.
- If the initial database state and all transactions are consistent, all consistency constraints are satisfied and the final database state will accurately reflect the real-world state, *but*
- Serial execution is inadequate from a performance perspective

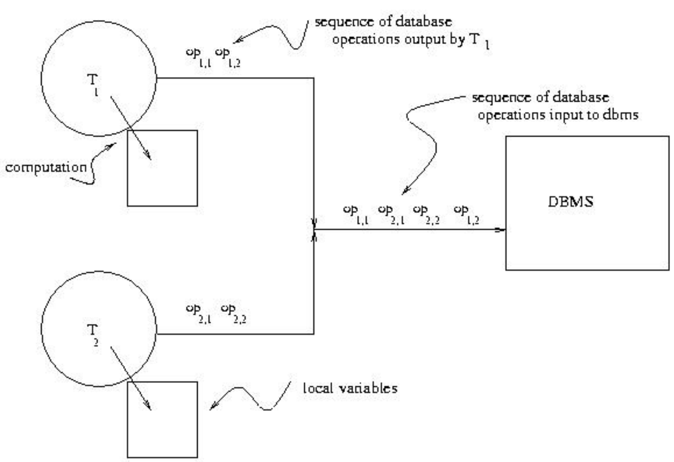
ACID

Isolation

- Concurrent execution offers performance benefits:
  - A computer system has multiple resources capable of executing independently (*e.g.*, cpu's, I/O devices), *but*
  - A transaction typically uses only one resource at a time
  - Concurrently executing transactions can make effective use of the system

ACID

Concurrent Execution



Example

- Consider two transactions:

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- **However, the net effect *must* be equivalent to these two transactions running serially in some order.**

Example (Contd.)

- Consider a possible interleaving (*schedule*):

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

- This is OK. But what about:

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

- The DBMS's view of the second schedule:

T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)

T2, T1: T1	T2, T1: T2	T1, T2: T1	T1, T2: T2	Missing interest Problem: T1	Missing interest Problem: T2	Interest for $100 twice Problem: T1	Interest for $100 twice Problem: T2
	Read(A)	Read(A)		Read(A)			Read(A)
	A=A*1.06	A=A+100		A=A+100			A=A*1.06
	Write(A)	Write(A)		Write(A)			Write(A)
Read(A)			Read(A)		Read(A)	Read(A)	
A=A+100			A=A*1.06		A=A*1.06	A=A+100	
Write(A)			Write(A)		Write(A)	Write(A)	
	Read(B)	Read(B)			Read(B)	Read(B)	
	B=B*1.06	B=B-100			B=B*1.06	B=B-100	
	Write(B)	Write(B)			Write(B)	Write(B)	
Read(B)			Read(B)	Read(B)			Read(B)
B=B-100			B=B*1.06	B=B-100			B=B*1.06
Write(B)			Write(B)	Write(B)			Write(B)

The net effect of an interleaved execution of T1 and T2 must be equivalent to the effect of running T1 and T2 in some serial order!

ACID

Isolation

- Concurrent (interleaved) execution of a set of consistent transactions offers performance benefits, *but* might not be correct
- **Example**: course registration; *cur_reg* is number of current registrants

T1: r(cur_reg : 29) w(cur_reg : 30)
T2: r(cur_reg : 29) w(cur_reg : 30)

time →

Result: Database state no longer corresponds to real-world state, integrity constraint violated (*cur_reg* <> |*list_of_registered_students*|)

ACID

Interaction of Atomicity and Isolation

T1: r(bal:10) w(bal:1000010) abort
T2: r(bal:1000010) w(yes!!!) commit

time →

- *T1* deposits $1000000
- *T2* grants credit and commits before *T1* completes
- *T1* aborts and rolls balance back to $10
- *T1* has had an effect even though it aborted!

ACID

Isolation

- An interleaved schedule of transactions is **isolated** if its effect is the same as if the transactions had executed serially in some order (**serializable**)
- It follows that serializable schedules are always correct (for any application)
- Serializable is better than serial from a performance point of view

ACID

Isolation in the Real World

- SQL supports SERIALIZABLE isolation level, which guarantees serializability and hence correctness for all applications
- Performance of applications running at SERIALIZABLE is often not adequate
- SQL also supports weaker levels of isolation with better performance characteristics
  - *But beware!* -- a particular application *might* not run correctly at a weaker level

ACID

Database Consistency

- **Enterprise (Business) Rules** limit the occurrence of certain real-world events
  - Student cannot register for a course if the current number of registrants equals the maximum allowed
- Correspondingly, allowable database states are restricted

  cur_reg <= *max_reg*
- These limitations are called (static) **integrity constraints***:* assertions that must be satisfied by the database state

ACID

Database Consistency

- Other static consistency requirements are related to the fact that the database might store the same information in different ways
  - *cur_reg* = |*list_of_registered_students*|
  - Such limitations are also expressed as integrity constraints
- **Database is consistent** if all static integrity constraints are satisfied

ACID Transaction Consistency

- A consistent database state does not necessarily model the actual state of the enterprise
  - A deposit transaction that increments the balance by the wrong amount maintains the integrity constraint *balance* ≥ 0, but does not maintain the relation between the enterprise and database states
- A consistent transaction maintains database consistency *and* the correspondence between the database state and the enterprise state (implements its specification)
  - Specification of deposit transaction includes
    balance = balance´+ amt_deposit ,
    (*balance´* is the initial value of *balance*)

ACID Dynamic Integrity Constraints

- Some constraints restrict allowable state transitions
  - A transaction might transform the database from one consistent state to another, but the transition might not be permissible
  - **Example**: A letter grade in a course (A, B, C, D, F) cannot be changed to an incomplete (I)
- Dynamic constraints cannot be checked by examining the database state

ACID Transaction Consistency

- A **transaction is consistent** if, assuming the database is in a consistent state initially, when the transaction completes:
  - All static integrity constraints are satisfied (but constraints might be violated in intermediate states)
    - Can be checked by examining snapshot of database
  - New state satisfies specifications of transaction
    - Cannot be checked from database snapshot
  - No dynamic constraints have been violated
    - Cannot be checked from database snapshot

ACID Checking Integrity Constraints

- Automatic: Embed constraint in schema.
  - CHECK, ASSERTION for static constraints
  - TRIGGER for dynamic constraints
  - Increases confidence in correctness and decreases maintenance costs
  - Not always desirable since unnecessary checking (overhead) might result
    - Deposit transaction modifies *balance* but cannot violate constraint *balance* ≥ 0
- Manual: Perform check in application code.
  - Only necessary checks are performed
  - Scatters references to constraint throughout application
  - Difficult to maintain as transactions are modified/added

ACID

Atomicity

- A real-world event either happens or does not happen
  - Student either registers or does not register
- Similarly, the system must ensure that either the corresponding transaction runs to completion or, if not, it has no effect at all
  - Not true of ordinary programs. A crash could leave files partially updated on recovery

ACID

Commit and Abort

- If the transaction successfully completes it is said to **commit**
  - The system is responsible for preserving the transaction's results in spite of subsequent failures
- If the transaction does not successfully complete, it is said to **abort**
  - The system is responsible for undoing, or **rolling back,** any changes the transaction has made

ACID

Reasons for Abort

- System crash
- Transaction aborted by system
  - Execution cannot be made atomic (a site is down)
  - Execution did not maintain database consistency (integrity constraint is violated)
  - Execution was not isolated
  - Resources not available (deadlock)
- Transaction requests to roll back

ACID

API for Transactions

- DBMS and TP monitor provide commands for setting transaction boundaries. For example:
  - begin transaction
  - commit
  - rollback
- The commit command is a request
  - The system might commit the transaction, or it might abort it for one of the reasons on the previous slide
- The rollback command is always satisfied

Summary

- Application programmer is responsible for creating consistent transactions
- DBMS and TP monitor are responsible for creating the abstractions of atomicity, durability, and isolation
  - Greatly simplifies programmer's task since he or she does not have to be concerned with failures or concurrency