CompSci 516
Data Intensive Computing Systems

Lecture 12
Intro to Transactions

Instructor: Sudeepa Roy

What will we learn?

- Last lecture:
  - Parallel DBMS and Map Reduce
  - Might be discussed more later for HW3 and HW4

- Next:
  - An introduction to Transactions
  - Will be continued for 4-5 lectures

Reading Material

- [RG]
  - Chapter 16.1-16.3, 16.4.1

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Motivation: Concurrent Execution

- Concurrent execution of user programs is essential for good DBMS performance.
  - Disk accesses are frequent, and relatively slow
  - it is important to keep the CPU busy by working on several user programs concurrently
  - short transactions may finish early if interleaved with long ones
  - may increase system throughput (avg. #transactions per unit time) and response time (avg time to complete a transaction)
- A user's program may carry out many operations on the data retrieved from the database
  - but the DBMS is only concerned about what data is read/written from/to the database.

Transactions

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

- A transaction is the DBMS's abstract view of a user program
  - a sequence of reads and write
  - the same program executed multiple times would be considered as different transactions
  - 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).

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

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

- 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)
```

Commit and Abort

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

- A transaction might commit after completing all its actions
- or it could abort (or be aborted by the DBMS) after executing some actions

Concurrency Control and Recovery

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

- Concurrency Control
  - (Multiple) users submit (multiple) transactions
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions
  - user should think of each transaction as executing by itself one-at-a-time
  - The DBMS needs to handle concurrent executions
- Recovery
  - Due to crashes, there can be partial transactions
  - DBMS needs to ensure that they are not visible to other transactions

ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity

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

- A user can think of a transaction as always executing all its actions in one step, or not executing any actions at all
  - Users do not have to worry about the effect of incomplete transactions
  - DBMS logs all actions so that it can undo the actions of aborted transactions.

Consistency

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

- Each transaction, when run by itself with no concurrent execution of other actions, must preserve the consistency of the database
  - e.g. if you transfer money from the savings account to the checking account, the total amount still remains the same
  - ensuring this property is the responsibility of the user

Isolation

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

- A user should be able to understand a transaction without considering the effect of any other concurrently running transaction
  - even if the DBMS interleaves their actions
  - transaction are "isolated or protected" from other transactions

Durability

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

- Once the DBMS informs the user that a transaction has been successfully completed, its effect should persist even if the system crashes before all its changes are reflected on disk

Durability

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

- Once the DBMS informs the user that a transaction has been successfully completed, its effect should persist even if the system crashes before all its changes are reflected on disk

Next, how we maintain all these four properties
But, in detail later

When can a transaction abort

- Transactions can be incomplete due to several reasons
  - Aborted (terminated) by the DBMS because of some anomalies during execution
    - in that case automatically restarted and executed anew
  - The system may crash (say no power supply)
  - A transaction may decide to abort itself encountering an unexpected situation
    - e.g. read an unexpected data value or unable to access disks

Atomicity and Durability

- Atomicity
  - A transaction interrupted in the middle can leave the database in an inconsistent state
  - DBMS has to remove the effects of partial transactions from the database
  - DBMS ensures atomicity by "undoing" the actions of incomplete transactions
  - DBMS maintains a "log" of all changes to do so
- Durability
  - The log also ensures durability
  - If the system crashes before the changes made by a completed transactions are written to the disk, the log is used to remember and restore these changes when the system restarts
  - "recovery manager" will be discussed later

Consistency and Isolation

- Consistency
  - e.g. Money debit and credit between accounts
  - User's responsibility to maintain the integrity constraints
  - DBMS may not be able to catch such errors in user program's logic
  - However, the DBMS may be in inconsistent state during a transaction between actions

- Isolation
  - DBMS guarantees isolation (later, how)
  - If T1 and T2 are executed concurrently, either the effect would be T1->T2 or T2->T1 (and from a consistent state to a consistent state)
  - But DBMS provides no guarantee on which of these order is chosen

Notations

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

- Transaction is a list of "actions" to the DBMS
  - includes "reads" and "writes"
  - $R_T(O)$: Reading an object O by transaction T
  - $W_T(O)$: Writing an object O by transaction T
  - also should specify $Commit_T$ and $Abort_T$
  - T is omitted if the transaction is clear from the context

Assumptions

- Transactions communicate only through READ and WRITE
  – i.e. no exchange of message among them

- A database is a fixed collection of independent objects
  – i.e. objects are not added to or deleted from the database
  – this assumption can be relaxed

Schedule

- An actual or potential sequence for executing actions as seen by the DBMS
- A list of actions from a set of transactions
  - includes READ, WRITE, ABORT, COMMIT
- Two actions from the same transaction T MUST appear in the schedule in the same order that they appear in T

Serial Schedule

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

- If the actions of different transactions are not interleaved
  - transactions are executed from start to finish one by one

Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions

- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

- Serializable schedule: A schedule that is equivalent to some serial execution of the committed transactions

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Serializable Schedule

- If the effect on any consistent database instance is guaranteed to be identical that of "some" complete serial schedule for a set of "committed trs"
- However, no guarantee on T1-> T2 or T2 -> T1

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

serial schedule

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	COMMIT
COMMIT	

T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	
	COMMIT
COMMIT	

serializable schedules

Anomalies with Interleaved Execution

- If two consistency-preserving transactions when run interleaved on a consistent database might leave it in inconsistent state

- Write-Read (WR)
- Read-Write (RW)
- Write-Write (WW)

- No conflict with RR if no write is involved

WR Conflict

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

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

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):
  - transaction T2 reads an object that has been modified by T1 but not yet committed
  - or T2 reads an object from an inconsistent database state (like fund is being transferred between two accounts)

RW Conflict

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

- Unrepeatable Reads (RW Conflicts):
  - T2 changes the value of an object A that has been read by transaction T1, which is still in progress
  - If T1 tries to read A again, it will get a different result
  - Suppose two customers are trying to buy the last copy of a book simultaneously

WW conflict

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

- Overwriting Uncommitted Data (WW Conflicts, "lost update"):
  - T2 overwrites the value of A, which has been modified by T1, still in progress
  - Suppose we need the salaries of two employees (A and B) to be the same
    - T1 sets them to $1000
    - T2 sets them to $2000

Schedules with Aborts

T1:	R(A), W(A),			Abort
T2:		R(A), W(A)	Commit	

- Actions of aborted transactions have to be undone completely
  - may be impossible in some situations
    - say T2 reads the fund from an account and adds interest
    - T1 aims to deposit money but aborts
  - if T2 has not committed, we can "cascade" aborts by aborting T2 as well
  - if T2 has committed, we have an "unrecoverable schedule"

Recoverable Schedule

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

- Transaction commit if and only after all transactions they read have committed
  - avoids cascading aborts

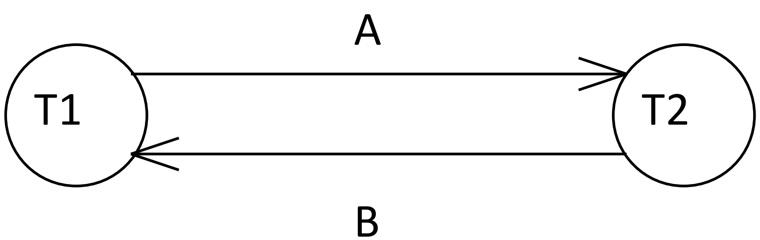
Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Example

- A schedule that is not conflict serializable:

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



Dependency graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Lock-Based Concurrency Control

- DBMS should ensure that only serializable and recoverable schedules are allowed
  - No actions of committed transactions are lost
- Uses a locking protocol

- Lock: associated with each “object”
  - different granularity

- Locking protocol:
  - a set of rules to be followed by each transaction

Strict two-phase locking (Strict 2PL)

Two rules

1. Each transaction must obtain
   - a S (*shared*) lock on object before reading
   - and an X (*exclusive*) lock on object before writing
   - exclusive locks also allow reading an object, additional shared lock is not required
   - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object
   - transaction is suspended until it acquires the required lock

2. All locks held by a transaction are released when the transaction completes

2PL vs. strict 2PL

- 2PL:
  - first, acquire all locks, release none
  - second, release locks, cannot acquire any other lock
- Strict 2PL:
  - release write (X) lock, only after it has ended (committed or aborted)
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - two transactions can acquire locks on different objects independently
- (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Example: Strict 2PL

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

- WR conflict (dirty read)
- Strict 2PL does not allow this

T1:	X(A), R(A), W(A),	
T2:		HAS TO WAIT FOR LOCK ON A

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

Example: Strict 2PL

T1:	S(A), R(A),		X(C), R(C), W(C), C
T2:		S(A), R(A), X(B), R(B), W(B), C	

- Strict 2PL allows interleaving

Transaction in SQL

- BEGIN TRANSACTION
- <.... SQL STATEMENTS>
- COMMIT or ROLLBACK

To be continued in the next lecture