CS 5614: (Big) Data Management Systems

B. Aditya Prakash

Lecture #6: Transactions 1: Intro. to ACID
Project dates

- Proposal due: Feb 23
- Milestone due: Mar 28
- Final report/posters etc: May 2 (last class)

- Check requirements for each online, especially proposal right now.
  - I will try to post some project ideas on Piazza by next week.
- Use the ‘Search for your teammates’ feature on Piazza to find partners.
Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.

- Unlike operating systems, which support interaction of processes, a DMBS needs to keep processes from troublesome interactions.
Transactions - dfn

- = unit of work, eg.
  - move $10 from savings to checking
Statement of Problem

- Concurrent execution of independent transactions (why do we want that?)
Statement of Problem

- Concurrent execution of independent transactions
  - utilization/throughput ("hide" waiting for I/Os.)
  - response time
Statement of Problem

- Concurrent execution of independent transactions
  - utilization/throughput ("hide" waiting for I/Os.)
  - response time
- would also like:
  - correctness &
  - fairness
- Example: Book an airplane seat
Definitions

- **database** - a fixed set of named data objects \((A, B, C, \ldots)\)

- **transaction** - a sequence of read and write operations \((\text{read}(A), \text{write}(B), \ldots)\)
  
  – DBMS’s abstract view of a user program
Example: ‘Lost-update’ problem

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(N)</td>
<td>Read(N)</td>
</tr>
<tr>
<td></td>
<td>N=N-1</td>
<td>N=N-1</td>
</tr>
<tr>
<td></td>
<td>Write(N)</td>
<td>Write(N)</td>
</tr>
</tbody>
</table>
Statement of problem (cont.)

- Arbitrary interleaving can lead to
  - Temporary inconsistency (ok, unavoidable)
  - “Permanent” inconsistency (bad!)

- Need formal correctness criteria.
Example: Bad Interaction

- You and friend each take $100 from different ATMs at about the same time.
  - The DBMS better make sure one account deduction doesn’t get lost.

- **Compare**: An OS allows two people to edit a document at the same time. If both write, one’s changes get lost.
ACID Transactions

- **ACID transactions** are:
  - **Atomic**: Whole transaction or none is done.
  - **Consistent**: Database constraints preserved.
  - **Isolated**: It appears to the user as if only one process executes at a time.
  - **Durable**: Effects of a process survive a crash.

- **Optional**: weaker forms of transactions are often supported as well (like Google, Amazon system etc.): Recall NoSQL systems
The SQL statement COMMIT causes a transaction to complete.

- It’s database modifications are now permanent in the database.
ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
  - No effects on the database.
- Failures like division by 0 or a constraint violation can also cause rollback, even if the programmer does not request it.
Overview

- **ACID transactions** are:
  - **Atomic**: Whole transaction or none is done.
  - **Consistent**: Database constraints preserved.
  - **Isolated**: It appears to the user as if only one process executes at a time.
  - **Durable**: Effects of a process survive a crash.
Atomicity of Transactions

- Two possible outcomes of executing a transaction:
  - Xact might *commit* after completing all its actions
  - or it could *abort* (or be aborted by the DBMS) after executing some actions.

- DBMS guarantees that Xacts are *atomic*.
  - From user’s point of view: Xact always either executes all its actions, or executes no actions at all.
Transaction states

- active
- partially committed
- committed
- failed
- aborted
Mechanisms for Ensuring Atomicity

- What would you do?
Mechanisms for Ensuring Atomicity

- One approach: LOGGING
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
- ~ like black box in airplanes ...
Mechanisms for Ensuring Atomicity

- Logging used by all modern systems.
- Q: why?
Mechanisms for Ensuring Atomicity

- Logging used by all modern systems.
- Q: why?
- A:
  - audit trail &
  - efficiency reasons
- What other mechanism can you think of?
Mechanisms for Ensuring Atomicity

- Another approach: SHADOW PAGES
  – (not as popular)
Overview

- **ACID transactions** are:
  - **Atomic**: Whole transaction or none is done.
  - **Consistent**: Database constraints preserved.
  - **Isolated**: It appears to the user as if only one process executes at a time.
  - **Durable**: Effects of a process survive a crash.
Transaction Consistency

- “Database consistency” - data in DBMS is accurate in modeling real world and follows integrity constraints
Transaction Consistency

- “Transaction Consistency”: if DBMS consistent before Xact (running alone), it will be after also
- Transaction consistency: User’s responsibility
  - DBMS just checks IC

consistent database S1  transaction T  consistent database S2
Recall: Integrity constraints
   – must be true for DB to be considered consistent
Examples:
1. FOREIGN KEY R.sid REFERENCES S
2. ACCT-BAL >= 0
System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).

- Beyond this, DBMS does not understand the semantics of the data.
- e.g., it does not understand how interest on a bank account is computed

Since it is the user’s responsibility, we don’t discuss it further
Overview

- **ACID transactions** are:
  
  - Atomic : Whole transaction or none is done.
  
  - Consistent : Database constraints preserved.
  
  - Isolated : It appears to the user as if only one process executes at a time.
  
  - Durable : Effects of a process survive a crash.
Isolation of Transactions

- Users submit transactions, and
- Each transaction executes as if it was running by itself.
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Q: How would you achieve that?
  - Tough problem!
Isolation of Transactions

- A: Many methods - two main categories:
  - Pessimistic – don’t let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them after they happen.
Example

- Consider two transactions (Xacts):

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

- 1st xact transfers $100 from B’s account to A’s
- 2nd credits both accounts with 6% interest.
- Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2?
Example

T1:
BEGIN
A=A+100, B=B-100
END

T2:
BEGIN
A=1.06*A, B=1.06*B
END

- many - but $A+B$ should be: $2000 \times 1.06 = 2120$
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect *must* be equivalent to these two transactions running serially in some order.
Example (Contd.)

- Legal outcomes: $A=1166, B=954$ or $A=1160, B=960$
- Consider a possible interleaved schedule:

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

  
- This is OK (same as T1;T2). But what about:

  | T1:  | A=A+100,       | B=B-100       |
  | T2:  | A=1.06*A,      | B=1.06*B      |
Example (Contd.)

- Legal outcomes: A=1166, B=954 or A=1160, B=960
- Consider a possible interleaved schedule:

<table>
<thead>
<tr>
<th>T1:</th>
<th></th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100,</td>
<td></td>
<td>A=1.06*A,</td>
</tr>
<tr>
<td>B=B-100</td>
<td></td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

- This is OK (same as T1;T2). But what about:

<table>
<thead>
<tr>
<th>T1:</th>
<th></th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100,</td>
<td></td>
<td>A=1.06*A,</td>
</tr>
<tr>
<td>B=B-100</td>
<td></td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

- Result: A=1166, B=960; A+B = 2126, bank loses $6

- The DBMS’ s view of the second schedule:

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A), R(B), W(B)</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>
‘Correctness’?

- Q: How would you judge that a schedule is ‘correct’?
  (‘schedule’ = ‘interleaved execution’)

Prakash 2017
‘Correctness’?

- Q: How would you judge that a schedule is ‘correct’?
- A: if it is equivalent to some serial execution
Serial schedule: Schedule that does not interleave the actions of different transactions.

Equivalent schedules: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule. (*)

(*) no matter what the arithmetic etc. operations are!
Formal Properties of Schedules

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

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
Anomalies with interleaved execution:

- R-W conflicts
- W-R conflicts
- W-W conflicts

(why not R-R conflicts?)
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

  T1: R(A), W(A), R(B), W(B), Abort
  T2: R(A), W(A), C
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

<table>
<thead>
<tr>
<th>T1: R(A), W(A), R(B), W(B), Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), C</td>
</tr>
</tbody>
</table>
Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

<table>
<thead>
<tr>
<th>T1: R(A), R(A), W(A), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), C</td>
</tr>
</tbody>
</table>

Prakash 2017

VT CS 5614
Anomalies with Interleaved Execution

- Unrepeatable Reads (RW Conflicts):

| T1:  | R(A), R(A), W(A), C |
| T2:  | R(A), W(A), C |
Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

| T1:  W(A), W(B), C |
| T2:  W(A), W(B), C |
Overwriting Uncommitted Data (WW Conflicts):

- T1: \( W(A), W(B), C \)
- T2: \( W(A), W(B), C \)
Serializability

- Objective: find non-serial schedules, which allow transactions to execute concurrently without interfering, thereby producing a DB state that could be produced by a serial execution

- BUT
  - Trying to find schedules equivalent to serial execution is too slow!
Conflict Serializability

- We need a formal notion of equivalence that can be implemented efficiently...
  - Base it on the notion of “conflicting” operations

- Definition: Two operations conflict if:
  - They are by different transactions,
  - they are on the same object,
  - and at least one of them is a write.
Conflict Serializable Schedules

- Definition: Two schedules are conflict equivalent iff:
  - They involve the same actions of the same transactions, and
  - every pair of conflicting actions is ordered the same way

- Definition: Schedule S is conflict serializable if:
  - S is conflict equivalent to some serial schedule.

- Note, some “serializable” schedules are NOT conflict serializable (See Example 4 later)
A schedule $S$ is conflict serializable if:
- You are able to transform $S$ into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

\[
\begin{align*}
\text{R(A) W(A)} & \quad \text{R(B) W(B)} \\
\text{R(A) W(A)} & \quad \text{R(B) W(B)} \\
\text{R(A) W(A) R(B) W(B)} & \quad \text{R(A) W(A) R(B) W(B)}
\end{align*}
\]
A schedule S is conflict serializable if:

- You are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions

R(A) W(A)

R(A) W(A) IS NOT SERIALIZABLE!
Serializability

- Q: any faster algorithm? (faster than transposing operations?)
Dependency Graph

- One node per Xact
- Edge from Ti to Tj if:
  - An operation Oi of Ti conflicts with an operation Oj of Tj and
  - Oi appears earlier in the schedule than Oj.
### Dependency Graph: Theorem

- **THEOREM:** Schedule is conflict serializable iff the dependency graph is acyclic

- Dependency graph is also called the precedence graph
  - different than the waits-for graph we will see later
Example

- T1: R(A), W(A)  \quad R(B), W(B)
- T2:  \quad R(A) W(A) R(B) W(B)

D. Graph:

- NOT Conflict serializable
  - Cycle is the problem---output of T1 depends on T2 and vice versa
Example #2 (Lost update)

\[ \text{T1} \]
Read(N)
N = N - 1
Write(N)

\[ \text{T2} \]
Read(N)
N = N - 1
Write(N)
Example #2 (Lost update)

T1

- Read(N)
- N = N - 1
- Write(N)

T2

- Read(N)
- N = N - 1
- Write(N)
Example #2 (Lost update)

T1
Read(N)
N = N - 1
Write(N)

T2
Read(N)
N = N - 1
Write(N)
Example #2 (Lost update)

T1

- **Read**($N$)
- $N = N - 1$
- **Write**($N$)

T2

- **Read**($N$)
- $N = N - 1$
- **Write**($N$)

T1 and T2 both perform **Read**($N$) and **Write**($N$) operations. However, due to the concurrent access, the expected outcome is that $N$ should be decremented twice, but due to the race condition, the update is lost.

R/W: Race condition between T1 and T2.
Example #3

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A)</td>
<td>Read(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example #3

T1  T2  T3
Read(A)  
...  
write(A)  
Read(A)  
...  
Write(A)  
Read(B)  
...  
Write(B)  
Read(B)  
...  
Write(B)

equivalent serial execution?
Example #3

- A: T2, T1, T3

(Notice that T3 should go after T2 in the equivalent serial order, although it starts before it!)

- Q: algo for generating serial execution from (acyclic) dependency graph?
Example #3

A: T2, T1, T3
(Notice that T3 should go after T2 in the equivalent serial order, although it starts before it!)

Q: algo for generating serial execution from (acyclic) dependency graph?
A: Topological sorting
Example #4 (Inconsistent Analysis)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R (A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>A = A-10</td>
<td>Sum = A</td>
</tr>
<tr>
<td>W (A)</td>
<td>R (B)</td>
</tr>
<tr>
<td></td>
<td>Sum += B</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>B = B+10</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>

dependency graph?
## Example #4 (Inconsistent Analysis)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R (A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>A = A-10</td>
<td>Sum = A</td>
</tr>
<tr>
<td>W (A)</td>
<td>R (B)</td>
</tr>
<tr>
<td></td>
<td>Sum += B</td>
</tr>
<tr>
<td></td>
<td>R (B)</td>
</tr>
<tr>
<td></td>
<td>B = B+10</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
</tbody>
</table>

So NOT Conflict Serializable (and not serializable)

dependency graph?
### Example #4 (Inconsistent Analysis)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R (A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>A = A - 10</td>
<td>Sum = A</td>
</tr>
<tr>
<td>W (A)</td>
<td>R (B)</td>
</tr>
<tr>
<td></td>
<td>Sum += B</td>
</tr>
<tr>
<td>R (B)</td>
<td></td>
</tr>
<tr>
<td>B = B + 10</td>
<td></td>
</tr>
<tr>
<td>W (B)</td>
<td></td>
</tr>
</tbody>
</table>

Q: create a ‘correct’ Schedule based on this one that is not conflict-serializable
Example #4’ (Inconsistent Analysis)

T1 
R (A)  
A = A-10  
W (A)

T2
R(A)  
if (A>0), count=1
R (B)  
if (B>0), count++
R(B)

B = B+10  
W(B)

A: T2 asks for the count of my active Accounts (assuming A>10, B>0)

NOTES:
1. This schedule is still not CS
2. BUT it is serializable! It is equivalent to either of [T1 T2] or [T2 T1] (both are OK)
Serializability in Practice

- DBMS does not test for conflict serializability of a given schedule
  - Impractical as interleaving of operations from concurrent Xacts could be dictated by the OS

- Approach:
  - Use specific protocols that are known to produce conflict serializable schedules
  - But may reduce concurrency
Solution?

- One solution for “conflict serializable” schedules is Two Phase Locking (2PL)
Answer

- (Full answer:) use locks; keep them until commit (‘strict 2 phase locking’)
- We’ll see the details later (in next class!)
(Review) Goal: ACID Properties

- **ACID transactions** are:
  - *Atomic*: Whole transaction or none is done.
  - *Consistent*: Database constraints preserved.
  - *Isolated*: It appears to the user as if only one process executes at a time.
  - *Durable*: Effects of a process survive a crash.

What happens if system crashes between *commit* and *flushing modified data to disk*?
Durability

- == Recovery
- We’ll see it later (after concurrency control)
Summary

- **Concurrency control and recovery** are among the most important functions provided by a DBMS.

- Concurrency control is automatic
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts
  - **Property ensured**: resulting execution is equivalent to executing the Xacts one after the other in some order.
ACID properties

Atomicity (all or none)
Consistency
Isolation (as if alone)
Durability