CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #20: Transactions
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
Transactions

- *Transaction* = process involving database queries and/or modification.
- Normally with some strong properties regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.
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
Example: ‘Lost-update’ problem

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

T2
Read(N)
N = N - 1
Write(N)
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.)
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.
Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- Only one level ("serializable") = ACID transactions.
- Each DBMS implements transactions in its own way.
Definitions

- A program may carry out many operations on the data retrieved from the database.
- However, the DBMS is only concerned about what data is read/written from/to the database.
Definitions

- database - a fixed set of named data objects (A, B, C, ...)
- transaction - a sequence of read and write operations (read(A), write(B), ...)
  – DBMS’s abstract view of a user program
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.
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)
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

\[
\text{consistent database S1} \xrightarrow{\text{transaction T}} \text{consistent database S2}
\]
Transaction Consistency (cont.)

- 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
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?
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):
  
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: BEGIN A=A+100, B=B-100 END</td>
<td></td>
</tr>
<tr>
<td>T2: BEGIN A=1.06<em>A, B=1.06</em>B END</td>
<td></td>
</tr>
</tbody>
</table>

- 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

\[
\begin{align*}
\text{T1: } & \text{ BEGIN } A=A+100, \ B=B-100 \text{ END} \\
\text{T2: } & \text{ BEGIN } A=1.06*A, \ B=1.06*B \text{ END}
\end{align*}
\]

- 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:

\[
\begin{array}{ll}
T1: & A = A + 100, \quad B = B - 100 \\
T2: & A = 1.06 \times A, \quad B = 1.06 \times B \\
\end{array}
\]

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

\[
\begin{array}{ll}
T1: & A = A + 100, \quad B = B - 100 \\
T2: & A = 1.06 \times A, \quad B = 1.06 \times B \\
\end{array}
\]
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:  | $A = 1.06 \times A$ | $B = 1.06 \times B$ |

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

| T1:  | $A = A + 100$ | $B = B - 100$ |
| T2:  | $A = 1.06 \times A$, $B = 1.06 \times B$ |

- Result: $A=1166$, $B=960$; $A+B = 2126$, bank loses $6$
- The DBMS’ s view of the second schedule:

| T1:  | $R(A), W(A)$, $R(B), W(B)$ |
| T2:  | $R(A), W(A)$, $R(B), W(B)$ |
Correctness’?

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

- Q: How would you judge that a schedule is ‘correct’?
- A: if it is equivalent to some serial execution
Formal Properties of Schedules

- 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”):

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

Prakash 2013
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>
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):

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

Prakash 2013
Anomalies (Continued)

- 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
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.

\[
\begin{align*}
R(A) & \quad W(A) & R(B) & \quad W(B) \\
R(A) & \quad W(A) & R(B) & \quad W(B) \\
R(A) & \quad W(A) & R(B) & \quad W(B) \\
R(A) & \quad W(A) & R(B) & \quad W(B)
\end{align*}
\]
CS---Intuition

- 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

$R(A) \quad W(A)$

$R(A) \quad W(A)$

*IS NOT SERIALIZABLE!*
Dependency Graph

- Dependency Graph
  - One node per Xact
  - Edge from Ti to Tj is Tj reads/writes an object last written by Ti

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

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

- D. Graph:

\[
\text{T1} \quad \text{Tw}
\]

- NOT Conflict serializable
  - Cycle is the problem---output of T1 depends on T2 and vice versa
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’)
- Let’s see the details
# Lost update problem - no locks

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read(N)</strong></td>
<td><strong>Read(N)</strong></td>
</tr>
<tr>
<td><strong>N = N - 1</strong></td>
<td><strong>N = N - 1</strong></td>
</tr>
<tr>
<td><strong>Write(N)</strong></td>
<td><strong>Write(N)</strong></td>
</tr>
</tbody>
</table>

Read(N)

N = N - 1

Write(N)
Solution – part 1

- with locks:
- lock manager: grants/denies lock requests
Lost update problem – with locks

T1

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

T2

lock manager

grants lock

denies lock

T2: waits

grants lock to T2

Read(N)
Locks

Q: I just need to read ‘N’ - should I still get a lock?
Solution – part 1

- Locks and their flavors
  - exclusive (or write-) locks
  - shared (or read-) locks
  - <and more ... >

- compatibility matrix

<table>
<thead>
<tr>
<th>T2 wants</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 has</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution – part 1

- Locks and their flavors
  - exclusive (or write-) locks
  - shared (or read-) locks
  - <and more ... >

- compatibility matrix

```
<table>
<thead>
<tr>
<th>T2 wants</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 has</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Solution – part 1

- transactions request locks (or upgrades)
- lock manager grants or blocks requests
- transactions release locks
- lock manager updates lock-table
Solution – part 2

locks are not enough – eg., the ‘inconsistent analysis’ problem
‘Inconsistent analysis’

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>A = A - 10</td>
<td>Sum = A</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>Sum += B</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B = B + 10</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>
‘Inconsistent analysis’ – w/ locks

<table>
<thead>
<tr>
<th>time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L(A)</td>
<td>L(A)</td>
</tr>
<tr>
<td></td>
<td>Read(A)</td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>U(A)</td>
<td>L(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>....</td>
</tr>
</tbody>
</table>

the problem remains!

T2 reads an inconsistent DB state

Solution??
General solution:

- Protocol(s)
- Most popular protocol: 2 Phase Locking (2PL)
2PL

X-lock version: transactions issue no lock requests, after the first ‘unlock’

THEOREM: if all transactions obey 2PL -> all schedules are serializable
2PL – example

- ‘inconsistent analysis’ – how does 2PL help?
- how would it be under 2PL?
transactions issue no lock/upgrade request, after the first unlock/downgrade

In general: ‘growing’ and ‘shrinking’ phase
transactions issue no lock/upgrade request, after the first unlock/downgrade

In general: ‘growing’ and ‘shrinking’ phase

violation of 2PL
2PL – observations

- limits concurrency
- may lead to deadlocks
  - We won’t cover in class
- strict 2PL (a.k.a. 2PLC): keep locks until ‘commit’
  - avoids ‘dirty reads’ etc
  - but limits concurrency even more
  - (and still may lead to deadlocks)
Aborting a Transaction (i.e., Rollback)

- If an xact $T_i$ aborted, all actions must be undone.
- On ‘dirty reads’: cascading aborts
- strict 2PL: avoids ‘dirty reads’ (why?)
  - But 2PL does not avoid cascading aborts

| T1: R(A), W(A), R(B), W(B), Abort |
| T2: R(A), W(A), |
Strict 2PL

- Allows only CS schedules, but it is actually stronger than needed for that purpose

Graph:
- Y-axis: # locks
- X-axis: time
- Growing phase: Release all locks at the end of the xact
Non-2PL, A= 1000, B=2000, Output =?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>A: = A-50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>Unlock(A)</td>
</tr>
<tr>
<td></td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>Unlock(B)</td>
</tr>
<tr>
<td></td>
<td>PRINT(A+B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(A)</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A: = A-50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_S(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B := B +50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>PRINT(A+B)</td>
<td></td>
</tr>
</tbody>
</table>

2PL, A= 1000, B=2000, Output =?
<table>
<thead>
<tr>
<th>Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A: = A-50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B := B +50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_S(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>PRINT(A+B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>

Strict 2PL, A= 1000, B=2000, Output =?
Venn diagram

- Serializable schedules
- 2PL schedules
- 2PLC
- Serial sch.’s
(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*?
Problem definition

- Records are on disk
- for updates, they are copied in memory
- and flushed back on disk, *at the discretion of the O.S.*! (unless forced-output: `output(B)’ = fflush())
Problem definition - eg.:

→ read(X)
X = X + 1
write(X)
Problem definition - eg.:

read(X)

→ X=X+1

write(X)
Problem definition - eg.:

read(X)
X=X+1
\[\rightarrow\] write(X)

buffer joins an output queue,
but it is NOT flushed immediately!

Q1: why not?

Q2: so what?
Problem definition - eg.:

- `read(X)`
- `read(Y)`
- `X = X + 1`
- `Y = Y - 1`
- `write(X)`
- `write(Y)`

Q2: so what?
Problem definition - eg.:

read(X)
read(Y)
X=X+1
Y=Y-1
write(X)
write(Y)

Q2: so what?
Q3: how to guard against it?
Solution: W.A.L.

- redundancy, namely
- write-ahead log, on ‘stable’ storage
- Q: what to replicate? (not the full page!!)
- A:
- Q: how exactly?
W.A.L. - intro

- replicate intentions: eg:

  <T1 start>
  <T1, X, 5, 6>
  <T1, Y, 4, 3>
  <T1 commit> (or <T1 abort>)
W.A.L. - intro

- in general: `<transaction-id, data-item-id, old-value, new-value>` (or similar)
- each transaction writes a log record first, **before** doing the change
- when done, DBMS
  - writes a `<commit>` record on the log
  - makes sure that all log records are flushed, &
  - lets xact exit
After a failure, DBMS “replays” the log:

- undo uncommitted transactions
- redo the committed ones
<T1 start>  
<T1, W, 1000, 2000>  
<T1, Z, 5, 10>  
<T1 commit>  

before crash

before

REDO T1

UNDO T1

Prakash 2013  
VT CS 4604
Logging (cont.)

- All logging and CC-related activities are handled transparently by the DBMS.
Durability - Recovering From a Crash

- At the end – all committed updates and only those updates are reflected in the database.
  - All active Xacts at time of crash are aborted when system comes back up.

- Some care must be taken to handle the case of a crash occurring during the recovery process!
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.
Summary

- Write-ahead logging (WAL) and the recovery protocol are used to:
  1. undo the actions of aborted transactions, and
  2. restore the system to a consistent state after a crash.
ACID properties

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