Lecture #20: Overview of Transaction Management (R&G ch. 16)

Components of a DBMS

Concurrency Control & Recovery

• Very valuable properties of DBMSs
• Based on concept of transactions with ACID properties
• Next lectures discuss these issues
Overview

- Problem definition & ‘ACID’
- Atomicity
- Consistency
- Isolation
- Durability

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

• would also like:
  – correctness &
  – fairness

• Example: Book an airplane seat

Example: ‘Lost-update’ problem

time

\begin{array}{|c|c|}
\hline
\text{Read(N)} & \text{Read(N)} \\
\text{N=N-1} & \text{N= N-1} \\
\text{Write(N)} & \text{Write(N)} \\
\hline
\end{array}
Statement of problem (cont.)

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

- Need formal correctness criteria.

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 \((A, B, C, \ldots)\)
- **transaction** - a sequence of read and write operations \((read(A), write(B), \ldots)\)
  - DBMS’s abstract view of a user program
Correctness criteria: The ACID properties

• Atomicity: All actions in the Xact happen, or none happen.
• Consistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
• Isolation: Execution of one Xact is isolated from that of other Xacts.
• Durability: If a Xact commits, its effects persist.

Overview

• Problem definition & ‘ACID’
  Atomicity
  Consistency
  Isolation
  Durability
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

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

- Problem definition & ‘ACID’
- Atomicity
- Consistency
  - Isolation
  - Durability

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

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

Transaction Consistency (cont.)

- 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

- Problem definition & ‘ACID’
- Atomicity
- Consistency
  * Isolation (‘as if alone’)
  * Durability

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):
  
  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 (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

• many - but A+B should be: $2000 *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:
  
  $T_1$: $A = A + 100$, $B = B - 100$  
  $T_2$: $A = 1.06A$, $B = 1.06B$
• This is OK (same as $T_1; T_2$). But what about:
  
  $T_1$: $A = A + 100$, $B = B - 100$  
  $T_2$: $A = 1.06A$, $B = 1.06B$
• Result: $A=1166$, $B=960$; $A+B = 2126$, bank loses $6$
• The DBMS’s view of the second schedule:
  
  $T_1$: $R(A)$, $W(A)$, $R(B)$, $W(B)$  
  $T_2$: $R(A)$, $W(A)$, $R(B)$, $W(B)$

‘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 e.t.c. operations are!

Serial 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

- 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: \(W(A), W(B), C\)

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

  T1: \(W(A), W(B), C\)
  T2: \(W(A), W(B), C\)
Solution?

• Q: How could you guarantee that all resulting schedules are correct (= serializable)?

Answer

• (Part of the answer:) use locks!

Answer

• (Full answer:) use locks; keep them until commit (‘strict 2 phase locking’)
  • Let’s see the details
Lost update problem - no locks

T1                        T2
Read(N)                   Read(N)
N = N - 1                 N = N - 1
Write(N)                  Write(N)

Solution – part 1

• with locks:
• lock manager: grants/denies lock requests

Lost update problem – with locks

T1                        T2
lock(N)                   lock manager
Read(N)                   grants lock
N = N - 1                 lock(N)  -  denies lock
Write(N)                  T2: waits
Unlock(N)                 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>S</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</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., ‘inconsistent analysis’

‘Inconsistent analysis’

<table>
<thead>
<tr>
<th>time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A)</td>
<td>A=A-10</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sum = A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B=B+10</td>
<td></td>
</tr>
<tr>
<td></td>
<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>Read(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>U(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the problem remains?
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?
- (answer: on the chalk-board)

2PL – X/S lock version

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

2PL – X/S lock version

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
- 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)

Abort ing a Transaction (i.e., Rollback)

- If an xact Ti aborted, all actions must be undone.
- On ‘dirty reads’: cascading aborts
- strict 2PL: avoids ‘dirty reads’ (why?)

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

Abort ing a Transaction (i.e., Rollback)

- To undo actions of an aborted transaction, DBMS maintains log which records every write.
- Log also used to recover from system crashes: All active Xacts at time of crash are aborted when system comes back up.
(Review) Goal: The ACID properties

- **Atomicity**: All actions in the Xact happen, or none happen.
- **Consistency**: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation**: Execution of one Xact is isolated from that of other Xacts.
- **Durability**: If a Xact commits, its effects persist.

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)

buffer

main memory

disk

Problem definition - eg.:

read(X)
→ X = X + 1
write(X)

main memory

disk

Problem definition - eg.:

read(X)
→ X = X + 1
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)
\rightarrow 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)
\rightarrow 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

W.A.L.

- After a failure, DBMS “replays” the log:
  - undo uncommitted transactions
  - redo the committed ones
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.
- Some care must be taken to handle the case of a crash occurring during the recovery process!
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.

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