CSC 742
Database Management Systems

Topic #12: Transaction Concepts
What is a transaction?

- Informally, a transaction is an execution of a program that satisfies certain properties.
  - Atomicity
  - Consistency
  - Isolation
  - Durability
Database Model for Transaction Processing

- A database is modeled as a collection of named items.
- Operations to database items are simplified to:
  - Read(x)
  - Write(x)
- The read-set (write-set) of a transaction is the set of items that the transaction reads (writes).
An Example Transaction

T1:

---------------------
Read(X)
X := X - 10
Write(X)
Read(Y)
Y := Y + 10
Write(Y)

Read-set (T1) = ?
Write-set (T1) = ?
What are transactions for?

- Concurrency control
- Recovery
Currency Control

- Currently running processes are actually interleaved.
- They may access same database items in these interleaved operations.
- The purpose of transactions (in the context of concurrency control) is to avoid the problems that may arise from interleaving
Problem 1: Lost Update

- The effect of a transaction on the DB is accidentally overwritten by another transaction

- $T_1 = r_1(a); a++; w_1(a)$

- $T_2 = r_2(a); a++; w_2(a)$
Problem 2: Dirty Read

A transaction prematurely reads a value from the DB that is later invalidated by another transaction.

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(X)</td>
<td>Read(X)</td>
</tr>
<tr>
<td></td>
<td>X := X – 10</td>
<td>X := X + 100</td>
</tr>
<tr>
<td></td>
<td>Write (X)</td>
<td>Write (X)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1 fails and has to change</td>
<td>Read(Y)</td>
<td></td>
</tr>
<tr>
<td>X back to its old value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T1 fails and has to change X back to its old value.
Problem 3: Incorrect Summary

Also termed *inconsistent analysis*
- A transaction observes an incorrect state of the DB
  - typically with some sort of aggregation

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(X)</td>
<td>Sum := 0</td>
</tr>
<tr>
<td>X := X - 10</td>
<td>Read (A)</td>
</tr>
<tr>
<td>Write (X)</td>
<td>Sum := Sum + A</td>
</tr>
<tr>
<td>Read (Y)</td>
<td>Read(X)</td>
</tr>
<tr>
<td>Y := Y + 100</td>
<td>Sum := Sum + X</td>
</tr>
<tr>
<td>Write (Y)</td>
<td>Read(Y)</td>
</tr>
<tr>
<td></td>
<td>Sum := Sum + Y</td>
</tr>
</tbody>
</table>

Time:
- T1
- T2

```
Read (X)
X := X – 10
Write (X)

Read (Y)
Y := Y + 100
Write (Y)
```
Recovery

- The DBMS needs to make sure that
  - Either all the operations of a transaction are completed and the effect is recorded permanently in the database,
  - Or the transaction has no effect at all.
- Recovery is necessary to make sure this is true in case of failures.
Desirable Transaction Properties

- These define traditional transactions
  - Atomicity or Failure Atomicity
    - All-or-nothing
    - If failed then no changes or messages
  - Consistency
    - Don't violate DB integrity constraints
  - Isolation or Isolation Atomicity
    - As if the transaction is being executed alone.
  - Durability
    - Effects (of transactions that "happened" or committed) are forever.
Atomicity or Failure Atomicity

- All or nothing
- If failed then
  - no changes
  - no messages
Consistency

- Don't violate DB integrity constraints
  - These constraints include
    - both the integrity constraints defined on database schemas and
    - other constraints that should hold on the database.
      - Rarely evaluated formally
      - Mostly just enforced by the programmer
Isolation or Isolation Atomicity

- Concurrent transactions are executed as if they are all executed alone.
- Partial results are hidden from users and other transactions
Durability

- Effects (of transactions that "happened" or committed) are forever.
- Should not affected by failures.
Programming with Transactions

- Transactions simplify programming
  - the programmer guarantees correctness of individual transactions
  - the system eventually recovers
  - then the system guarantees the ACID properties for the entire set of transactions that execute concurrently
Transaction States and Operations

- Begin Transaction
- Active
  - Read or Write
  - End Transaction
  - Partially Committed
    - Why partially committed?
    - Partially Committed
      - Abort
      - Commit Transaction
    - Committed
      - Terminated
- Failed
  - Abort

Why partially committed?
System Log

- System log:
  - To keep track of all transaction operations.
  - Kept on disk.
  - Necessary for recovery from failures.

- Log records: Entries in the system log
  - [Start_Transaction, T]
  - [Write, T, old_value, new_value]
  - [Read, T, X] (May not be required, depending on the protocol.)
  - [Commit, T]
  - [Abort, T]
  - [checkpoint]
Commit Point of a Transaction

- A transaction T reaches its *commit point* when
  - All its operations that access the database have been executed, and
  - The effect of all transaction operations have been recorded in the log.

- A transaction is *committed* beyond its commit point.
  - [Commit, T] in the log.

- *Force write* the log:
  - Before T’s commit point, all of the log must be written to the disk.
  - Why?
Checkpoint

- [Checkpoint] is appended to the log when writes of all previously committed transactions are effected on the database.

- Purpose:
  - Reduce the work required for recovery.
Schedules

- Schedules are histories of computations showing all the events of interest
  - Schedule or history of T1...Tn has operations of T1...Tn
  - Operations may be interleaved, but must be in the same order as within each Ti
  - Events of interest:
    - Read, Write, Commit, Abort in Ti
    - Abbreviations: r_i, w_i, c_i, a_i.
Read (X)
X := X – 10
Write (X)

Read (Y)

Read (X)
X := X + 100
Write (X)

Schedule:
Read (X)
X := X - 10
Write (X)

Read (Y)
Y := Y + 100
Write (Y)

Sum := 0
Read (A)
Sum := Sum + A

Read (X)
Sum := Sum + X
Read (Y)
Sum := Sum + Y

Schedule:
Conflict

- Two operations in a schedule conflict if
  - They are in different transactions
  - They involve the same data item
  - At least one of them is write.

What are the conflict operations?

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(X)</td>
<td>Read(X)</td>
</tr>
<tr>
<td></td>
<td>X := X - 10</td>
<td>X := X + 100</td>
</tr>
<tr>
<td></td>
<td>Write(X)</td>
<td>Write(X)</td>
</tr>
<tr>
<td></td>
<td>Read(Y)</td>
<td>Read(Y)</td>
</tr>
</tbody>
</table>
Complete Schedule

- A schedule S of n transactions T1, …, Tn is a complete schedule if
  - The operations in S are exactly those in T1, …, Tn, including a commit or abort as the last operation for each Ti;
  - For any pair of operations from the same Ti, their order is the same as in Ti;
  - For any two conflicting operations, one must occur before the other.

- A partial order in which
  - a commit or abort event for each Ti is included
  - conflicting ops are ordered
Examples

Transactions:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>r1(X); w1(X); r1(Y); w1(Y); c1;</td>
</tr>
<tr>
<td>T2:</td>
<td>r2(Y); w2(Y); a2;</td>
</tr>
<tr>
<td>T3:</td>
<td>r3(X); w3(X);</td>
</tr>
</tbody>
</table>

S1: r1(X); r2(Y); w1(X); w2(Y); r1(Y); a2; w1(Y); c1; r3(X); w3(X);

S2: r1(X); r2(Y); w1(X); w2(Y); r1(Y); a2; w1(Y); c1;

S3: r1(X); w2(Y); w1(X); r2(Y); r1(Y); a2; w1(Y); c1;

Is S1 a complete schedule? For what?
Committed Projection

- A committed projection: C(S) of a schedule S include the operations in S only from the committed transactions.

Transactions:

\[
\begin{align*}
T1: & \ r1(X); w1(X); r1(Y); w1(Y); c1; \\
T2: & \ r2(Y); w2(Y); a2; \\
T3: & \ r3(X); w3(X); \\
\end{align*}
\]

\[
\begin{align*}
S1: & \ r1(X); r2(Y); w1(X); w2(Y); r1(Y); a2; w1(Y); c1; r3(X); w3(X); \\
\end{align*}
\]

\[
C(S1) = ?
\]
Recoverable Schedules

- Goal:
  - Once a transaction T is committed, it should never be necessary to roll back T.
  - Recoverable schedules: schedules that meet this criterion.

\[ S1: r1(X); w1(X); r2(X); w2(X); c2; a1; \]

Is this recoverable?

In terms of the ACID properties, what is the risk in allowing a non-recoverable schedule?
Recoverable Schedules (Cont’d)

- A schedule S is *recoverable* if each transaction commits after all transactions it *read from* have committed

  - T reads from T2 if the schedule contains a subsequence \( w_2(x) \ldots r(x) \), where \( w_2 \) is the first write on \( x \) going backwards from \( r(x) \).

```plaintext
S1: r1(X); w1(X); r2(X); w2(X); c1; c2;
```
Avoid Cascading Aborts

- A schedule S can *avoid cascading aborts* if no transaction in S read from uncommitted transactions.

- What is the risk in doing so?

\[
\begin{align*}
S1: & \ r1(X); \ w1(X); \ r2(X); \ w2(X); \ a1; \ a2; \\
S2: & \ r1(X); \ w1(X); \ r2(X); \ w2(X); \ r3(X); \ w3(X); \ a1; \ a2; \ a3; \\
S3: & \ r1(X); \ w1(X); \ a1; \ r2(X); \ w2(X); \ c2; \ r3(X); \ w3(X); \ c3;
\end{align*}
\]
Strict Schedules

- A schedule S is *strict* if no transaction can read or write an item X until the previous transaction to write that item has committed or aborted.

\[
S1: r1(X); w1(X); w2(X); c1; c2;
\]

\[
S2: r1(X); w1(X); c1; w2(X); c2;
\]

Avoid Cascading Abort?
Strict?
Strict Schedules (Cont’d)

- Benefit:
  - Allows us to UNDO by restoring the before image

\[
\begin{align*}
X &= 9 \\
S1: &\ w1(X, 5); \ w2(X, 8); \ a1; \\
&\ X = 9 \\
Log: &\ \ldots \\
&\ [\text{write}, \ T1, \ X, \ 9, \ 5] \\
&\ [\text{write}, \ T2, \ X, \ 5, \ 8] \\
&\ [\text{abort}, \ T1] \\
&\ \ldots \\
&\ X = ?
\end{align*}
\]

\[
\begin{align*}
X &= 9 \\
S1: &\ w1(X, 5); \ a1; \ w2(X, 8); \\
&\ X = 9 \\
Log: &\ \ldots \\
&\ [\text{write}, \ T1, \ X, \ 9, \ 5] \\
&\ [\text{abort}, \ T1] \\
&\ [\text{write}, \ T2, \ X, \ 9, \ 8] \\
&\ \ldots \\
&\ X = ?
\end{align*}
\]
Note

- Strict $\rightarrow$ Avoid Cascading Abort $\rightarrow$ Recoverable.
- All of them are about recovery.
Serial Schedules

Transactions are wholly before or after others.
- Serial schedules are correct (assuming each transaction is correct)
- Clearly, we must allow for service requests to come in slowly, one-by-one.

```
Read(X)
X := X - 10
Write(X)
```

```
Read(X)
X := X + 100
Write(X)
```
Non-serial Schedules

\[ X = 10, \ Y = 20 \]

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read (X)</td>
<td>X := X – 10</td>
<td>Read (X)</td>
</tr>
<tr>
<td>Write (X)</td>
<td></td>
<td>Write (X)</td>
</tr>
<tr>
<td>Y := Y + 20</td>
<td></td>
<td>Y := Y + 20</td>
</tr>
<tr>
<td>Write (Y)</td>
<td></td>
<td>Write (Y)</td>
</tr>
</tbody>
</table>

Are they correct?

X = 10, Y = 20
Serializable Schedules: 1

- **Serializable schedules** are those equivalent to some serial schedule.
- Here equivalent can mean
  - **conflict equivalent**
    - The order of any two conflicting operations is the same in both schedules.
    - Alternatively, to be equivalent to a serial schedule, a non-serial schedule must have all pairs of conflicting operations ordered the same way.
  - **view equivalent**—each read of a transaction gets the same view in both schedules.
\[ X = 10, \ Y = 20 \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(X)</td>
<td>Read(X)</td>
</tr>
<tr>
<td>X := X − 10</td>
<td>X := X + 100</td>
</tr>
<tr>
<td>Write (X)</td>
<td>Write (X)</td>
</tr>
<tr>
<td>Read (Y)</td>
<td>Read (Y)</td>
</tr>
<tr>
<td>Y := Y + 20</td>
<td>Y := Y + 20</td>
</tr>
<tr>
<td>Write (Y)</td>
<td>Write (Y)</td>
</tr>
</tbody>
</table>

Order of conflicting operations?  
Are they conflict equivalent to any serial schedule?  
Are they serializable w.r.t. conflict?
Serializable Schedules: 2

- Most approaches use conflict equivalence, because conflict equivalence
  - entails view equivalence
  - is relatively easy to ensure
Checking Serializability

- Build a *serialization graph* for the given concurrent mix of transactions:
  - a node for each transaction
  - a directed edge for each conflict
  - serializability $\iff$ the graph is acyclic
S1: r2(Z); r2(Y); w2(Y); r3(Y); r3(Z); r1(X); w1(X); w3(Y); w3(Z); r2(X); r1(Y); w1(Y); w2(X);

S2: r3(Y); r3(Z); r1(X); w1(X); w3(Y); w3(Z); r2(Z); r1(Y); w1(Y); r2(Y); w2(Y); r2(X); w2(X);
Serializable Schedules: 3

Can we find an equivalent serial schedule for a serializable schedule?

- Such a serial schedule exists by definition
- Can easily derive from serialization graph.
Equivalent serial schedules: ?
Achieving Serializability

- Optimistically: Let each transaction run, but check for serializability before committing.
- Pessimistically: Use a protocol, e.g., locking, to ensure that only serializable schedules are realized

Generally, the pessimistic approach is more common