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
- T1 = r1(a): a++; w1(a)
- T2 = r2(a): a++; w2(a)

Problem 2: Dirty Read
- A transaction prematurely reads a value from the DB that is later invalidated by another transaction
- 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

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

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.
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 be 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

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.
  - 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, w, c, a.

T1 T2

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

Schedule:

Conflicts

- 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.
Complete Schedule

- A schedule $S$ of $n$ transactions $T_1, \ldots, T_n$ is a complete schedule if
  - The operations in $S$ are exactly those in $T_1, \ldots, T_n$, including a commit or abort as the last operation for each $T_i$;
  - For any pair of operations from the same $T_i$, their order is the same as in $T_i$;
  - For any two conflicting operations, one must occur before the other.
- A partial order in which
  - a commit or abort event for each $T_i$ is included
  - conflicting ops are ordered

Examples

Transactions:
$$T_1: r(X); w(X); r(Y); w(Y); c_1; T_2: r(X); w(Y), a_2; T_3: r(X); w(X);$$

$S_1: r(X); r(Y); w(X), r(Y), a_2; w(Y); c_1; r(X); w(X)$

Is $S_1$ 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:
$$T_1: r(X); w(X); r(Y); w(Y); c_1; T_2: r(Y); w(X), a_2; T_3: r(Y); w(X);$$

$S_1: r(X); r(Y); w(X), r(Y), a_2; w(Y); c_1; r(Y); w(X)$

$C(S_1) =$ ?

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.

Transactions:
$$S_1: r(X); w(X), c_1; w(Y); c_2; a_1;$$

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 $T_2$ if the schedule contains a subsequence $w_2(x)\ldots r(x)$, where $w_2$ is the first write on $x$ going backwards from $r(x)$.

Transactions:
$$S_1: r(X); w(X), r(X); w(X), c_1; c_2;$$

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?

Transactions:
$$S_1: r(X); w(X), r(X); w(X), a_1; a_2;$$

$S_2: r(X); w(X), r(X); w(X), a_1; a_2, a_3;$$

$S_3: r(X); w(X), a_1, r(X); w(X), c_3; r(X); w(X), c_3;$$
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.

$S_1: r(X); w(X); w(X), c_1, c_2$

$S_2: r(X); w(X), c_1, w(2X), c_2$

Avoid Cascading Abort? Strict?

Note

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

Strict Schedules (Cont’d)

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

<table>
<thead>
<tr>
<th>Log:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X=9$</td>
</tr>
<tr>
<td>$S_1: w(X, 5); w(2X, 8); a_1; \rightarrow$</td>
</tr>
<tr>
<td>$T_1, T_2, X, 5, 8$</td>
</tr>
<tr>
<td>$X=9$</td>
</tr>
<tr>
<td>$S_1: w(X, 5); a_1; w(2X, 8); \rightarrow$</td>
</tr>
<tr>
<td>$T_1, T_2, X, 5, 8$</td>
</tr>
</tbody>
</table>

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.

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.
  - **view equivalent**—each read of a transaction gets the same view in both schedules.

Non-serial Schedules

<table>
<thead>
<tr>
<th>$X = 10, Y = 20$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
</tr>
<tr>
<td>Time:</td>
</tr>
<tr>
<td>Read(X)</td>
</tr>
<tr>
<td>Write(X)</td>
</tr>
<tr>
<td>Read(Y)</td>
</tr>
<tr>
<td>Write(Y)</td>
</tr>
</tbody>
</table>

Are they correct?
Order of conflicting operations?
Are they conflict equivalent to any serial schedule?
Are they serializable w.r.t. conflict?

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

Serializable Schedules: 2
- Most approaches use conflict equivalence, because conflict equivalence
  - entails view equivalence
  - is relatively easy to ensure

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.
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.