Transactions

Serializability
Isolation Levels
Atomicity

Example: Bad Interaction
◆ You and your spouse each take $100 from different ATM’s 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 Test
◆ A DBMS is expected to support “ACID transactions,” which are:
  ◆ Atomic: Either the whole process is done or none is.
  ◆ Consistent: Database constraints are preserved.
  ◆ Isolated: It appears to the user as if only one process executes at a time. (aka “serializable”)
  ◆ Durable: Effects of a process do not get lost if the system crashes.

Transactions in SQL
◆ SQL supports transactions, often behind the scenes.
  ◆ Each statement issued at the generic query interface is a transaction by itself.
  ◆ In programming interfaces like Embedded SQL or PSM, a transaction begins the first time an SQL statement is executed and ends with the program or an explicit end.

COMMIT
◆ The SQL statement COMMIT causes a transaction to complete.
  ◆ Its 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 can also cause rollback, even if the programmer does not request it.

An Example: Interacting Processes

◆ Assume the usual Sells(bar,beer,price) relation, and suppose that Joe’s Bar sells only Bud for $2.50 and Miller for $3.00.
◆ Sally is querying Sells for the highest and lowest price Joe charges.
◆ Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50.

Sally’s Program

◆ Sally executes the following two SQL statements, which we call (min) and (max), to help remember what they do.
  (max) SELECT MAX(price) FROM Sells WHERE bar = 'Joe’s Bar';
  (min) SELECT MIN(price) FROM Sells WHERE bar = 'Joe’s Bar';

Joe’s Program

◆ At about the same time, Joe executes the following steps, which have the mnemonic names (del) and (ins).
  (del) DELETE FROM Sells WHERE bar = 'Joe’s Bar';
  (ins) INSERT INTO Sells VALUES('Joe’s Bar', 'Heineken', 3.50);

Interleaving of Statements

◆ Although (max) must come before (min) and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally’s and/or Joe’s statements into transactions.

Example: Strange Interleaving

◆ Suppose the steps execute in the order (max)(del)(ins)(min).
  Joe’s Prices: 2.50, 3.00 2.50, 3.00 3.50
  Statement: (max) (del) (ins) (min)
  Result: 3.00 3.50
◆ Sally sees MAX < MIN!
Fixing the Problem With Transactions

- If we group Sally’s statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She sees Joe’s prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

- Suppose Joe executes (del)(ins), but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her transaction after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database.

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can never be seen.

Isolation Levels

- SQL defines four isolation levels = choices about what interactions are allowed by transactions that execute at about the same time.
  - How a DBMS implements these isolation levels is highly complex, and a typical DBMS provides its own options.

Choosing the Isolation Level

- Within a transaction, we can say:
  SET TRANSACTION ISOLATION LEVEL X
  where X =
  1. SERIALIZABLE (strongest)
  2. REPEATABLE READ
  3. READ COMMITTED
  4. READ UNCOMMITTED (weakest, effectively unrestricted)

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.
  - It’s up to the DBMS vendor to figure out how to do that, e.g.:
    - True isolation in time.
    - Keep Joe’s old prices around to answer Sally’s queries.
Isolation Level Is “Personal” Choice

- Your choice, e.g., run serializable, affects only how you see the database, not how others see it.
- Example: If Joe runs serializable, but Sally doesn’t, then Sally might see no prices for Joe’s Bar.
  - i.e., it looks to Sally as if she ran in the middle of Joe’s transaction.

Read-Commited Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see committed data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits.
  - Sally sees MAX < MIN.

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see more tuples as well.

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
  - (max) sees prices 2.50 and 3.00.
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.

Locking

- Locking is a system-level method for achieving serialization.
- The diagrams in slides on locking are modifications of ones due to Arthur Keller, rather than Jeff Ullman, I believe.
Non-Serialized Transaction

- T1
  - start with A = 5
  - Read A
  - A := A + 1
  - Write A
- T2
  - Read A
  - A := 2 * A
  - Write A

Read- vs. Write-Locks

<table>
<thead>
<tr>
<th>THEM</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLOCK A</td>
<td>NO</td>
<td>OK</td>
</tr>
<tr>
<td>WLOCK A</td>
<td>US</td>
<td>R</td>
</tr>
<tr>
<td>UNLOCK A</td>
<td>W</td>
<td>OK</td>
</tr>
</tbody>
</table>

RLock -> UNLOCK can enclose a read
WLock -> UNLOCK can enclose a write or read

Transaction Serialized using Locks

T1
- WLock A
- Read A
- A := A + 1
- Write A
- UNLock A

T2
- WLock A

Deadlock due to lock ordering

T1
- WLock A
- WLock B
- Wait
- UNLock A

T2
- WLock B
- Request lock upgrade
- Wait
- UNLock B
- UNLock A

Recent A

A := A + 1

Red A

A := 2 * A

Write A

A := 2 * A

WLock A

Wait

Deadlock!

Problems with Unstructured Use of Locks

T1
- RLock A
- Read A
- A := A + 1
- WLock A
- Wait
- Wait

T2
- RLock A
- Read A
- A := 2 * A
- WLock A
- Request lock upgrade
- Wait

Wait-for graph

Deadlock

1. Wait and hold some locks while you wait for others
2. Circular chain of waiters
3. No pre-emption

We can avoid deadlock by doing at least ONE of:
1. Get all your locks at once
2. Apply an ordering to acquiring locks
3. Allow preemption (for example, use timeout on waits)
Serializability of schedules

<table>
<thead>
<tr>
<th>Schedule</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: Read (A)</td>
<td>T2: Write (A)</td>
<td>disk 100 200</td>
</tr>
<tr>
<td>A:= A-50</td>
<td>temp:= A * 0.1</td>
<td></td>
</tr>
<tr>
<td>Write (A)</td>
<td>Read (B)</td>
<td></td>
</tr>
<tr>
<td>B:= B+50</td>
<td>Write (B)</td>
<td></td>
</tr>
<tr>
<td>Write (B)</td>
<td>T1</td>
<td></td>
</tr>
</tbody>
</table>

A schedule is serializable if its effect is the same as some serial schedule.

T1 -> T2
A =
B =

Serializability Test

**Wanted:**

A way to test serializability by analyzing lock operations among transactions.

**Algorithm:** Testing serializability of a schedule

**Input:** Schedule S for transactions T1, ..., Tk

**Output:** Determination of whether S is serializable, and if so, an equivalent serial schedule.

**Method:**
1. Create a directed graph G (called a serialization graph)
2. Create a node for each transaction and label with transaction ID
3. Create an edge for each Ti: UNLOCK A followed by Tj: LOCK A (where lock modes conflict). (A is the data item being locked.)
4. Label edge Ti -> Tj with A.

If there is a cycle then schedule is non-serializable.
If there is no cycle, then (it is a DAG) do a topological sort to get a serial schedule

DAG implies some partial order. Any total order consistent with the partial order is an equivalent serial schedule.

Serializability Test Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Simplest Non-Serializable Case

\[
\begin{array}{c|c}
\text{T1} & \text{T2} \\
\hline
\text{WLOCK A} & \text{WLOCK A} \\
\text{UNLOCK A} & \text{WLOCK B} \\
\hline
\text{WLOCK B} & \text{UNLOCK A} \\
\text{UNLOCK B} & \text{UNLOCK B} \\
\end{array}
\]

Possible Serialization Strategies

\begin{itemize}
\item Static transaction analysis
\item Dynamic detection of non-serializability, rollback if necessary
\item Devise a safe but simple policy that avoids non-serializability
\end{itemize}

Simple Strategy: 2-Phase Locking (2PL)

Every transaction has is divided into two phases that occur in sequence:

- **Phase I**: All requesting of locks (no releasing)
- **Phase II**: All releasing of locks (no further requesting)

**Theorem**: Any schedule for 2-phase locked transaction is serializable

[Similar-sounding, but distinct idea: 2-Phase Commit used in distributed databases.]

Rollback Hazards

\[
\begin{array}{c|c}
\text{T1} & \text{T2} \\
\hline
\text{LOCK A} & \text{LOCK A} \\
\text{Read A} & \text{Read A} \\
\text{change A} & \text{change A} \\
\text{Write A} & \text{Write A} \\
\hline
\text{UNLOCK A} & \text{UNLOCK A} \\
\text{UNLOCK A} & \text{UNLOCK A} \\
\end{array}
\]

LOCK B
Read B
Discover problem
ABORT: ROLLBACK

Need to undo the change to A by T2 that was made based on data written by T1.
(CASCADED ROLLBACK)