Distributed Transactions

Presented By:
Mahmoud ElGammal
What is a Transaction?

- An execution of a program that accesses a shared database.
- Consists of an ordered set of operations that must be executed in sequence.
- Has to be executed atomically:
  - Each transaction accesses shared data without interfering with other transactions.
  - If a transaction terminates normally, then all of its effects are made permanent; otherwise it has no effect at all.
The Problem

- Due to their being non-atomic by nature, concurrent execution of transactions can cause *interference*.
- Interference can lead to *inconsistency*, rendering the DBMS unreliable for data storage!
- Solution: Concurrency control.
  - The activity of coordinating the actions of processes that operate in parallel, access shared data, and therefore potentially interfere with each other.
The Goal of Concurrency Control

Maximizing throughput without compromising consistency.

- Unsupervised concurrency achieves high throughput, but also causes high contention $\rightarrow$ Consistency is lost.
- Absence of concurrency achieves consistency, but also eliminates sharing $\rightarrow$ Worst throughput.
- We need to find a point in-between where we can achieve:
  - Concurrent execution.
  - Final effect same as some sequential execution.
Serializability

- The DBMS interleaves the execution of operations from different transactions.
- It doesn’t (and needn’t) make any promises about the order in which different transactions will execute relative to each other.
- We can broaden the class of allowable executions to include executions that have the same effect as a serial ones.
- Such executions are called *serializable*.
- Since they have the same effect as serial executions, serializable executions are correct.
Serializability

DB is consistent if it is guaranteed to have resulted from any one of:

\[
\begin{align*}
T_1 & \rightarrow T_2 & \rightarrow T_3 \\
T_2 & \rightarrow T_1 & \rightarrow T_3 \\
T_2 & \rightarrow T_3 & \rightarrow T_1 \\
T_1 & \rightarrow T_3 & \rightarrow T_2 \\
T_3 & \rightarrow T_1 & \rightarrow T_2 \\
T_3 & \rightarrow T_2 & \rightarrow T_1
\end{align*}
\]
Consider these two transactions:

\[ T_1: R_1(Z) \ W_1(Y) \ W_1(X) \quad T_2: W_2(Y) \ W_2(Z) \ R_2(X) \]

Is this execution log serializable?

\[ L: R_1(Z) \ W_1(Y) \ W_2(Y) \ W_2(Z) \ R_2(X) \ W_1(X) \]
Serializability

\[ S_1: R_1(Z) \ W_1(Y) \ W_1(X) \ W_2(Y) \ W_2(Z) \ R_2(X) \]

\[ L: R_1(Z) \ W_1(Y) \ W_2(Y) \ W_2(Z) \ R_2(X) \ W_1(X) \]

\[ S_2: W_2(Y) \ W_2(Z) \ R_2(X) \ R_1(Z) \ W_1(Y) \ W_1(X) \]

Read source conflict

Last write conflict
Scheduling

- The DBMS’s scheduler restricts the order in which transactional operations are executed.
- The goal is to order these operations so that the resulting execution is serializable.
- Each data item has a lock associated with it.
- The scheduler utilizes a protocol for executing, rejecting, or delaying an operation according to lock states.

<table>
<thead>
<tr>
<th>Lock Request</th>
<th>Not locked</th>
<th>READ locked</th>
<th>WRITE locked</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>OK</td>
<td>OK</td>
<td>DENY (=defer)</td>
</tr>
<tr>
<td>WRITE</td>
<td>OK</td>
<td>DENY (=defer)</td>
<td>DENY (=defer)</td>
</tr>
</tbody>
</table>
Two Phase Locking (2PL)

- 2PL is a locking protocol that guarantees serializability.
- Consists of three rules:
  - On receiving a conflicting operation, delay the requesting transaction until the requested lock is released then assign the lock to it.
  - A lock is never released until its associated operation is processed.
  - Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item).

![Diagram showing the locking and release phases of 2PL]

- locking phase
- release phase

- $T_i$:
  - no locks released
  - no new locks requested

- time
Two Phase Locking (2PL)

- Rule (1) prevents two transactions from concurrently accessing a data item in conflicting modes.
- Rule (2) ensures that the DM processes operations on a data item in the order that the scheduler submits them.
- Rule (3) guarantees that all pairs of conflicting operations of two transactions are scheduled in the same order.

\[
T_1: r_1[x] \rightarrow w_1[y] \rightarrow c_1 \quad T_2: w_2[x] \rightarrow w_2[y] \rightarrow c_2
\]

\[
L_1: rl_1[x] r_1[x] ru_1[x] w_2[x] w_2[x] w_2[y] w_2[y] w_2[x] w_2[y] c_2 w_1[y] w_1[y] w_1[y] c_1
\]

\[
L_2: rl_1[x] r_1[x] w_1[y] w_1[y] c_2 ru_1[x] w_1[y] w_1[y] w_1[y] w_1[y] w_1[y] w_1[y] c_2 w_2[x] w_2[x] w_2[y] w_2[y] c_2 w_2[x] w_2[y]
\]
Two Phase Locking (2PL)

- Deadlocks still can happen:
  \[ T_1: w_1[x] \rightarrow w_1[y] \rightarrow c_1 \quad T_2: w_2[y] \rightarrow w_2[x] \rightarrow c_2 \]

  \[ L_1: wl_1[x] \quad w_1[x] \quad wl_2[y] \quad w_2[y] \ldots ? \]

- The scheduler constructs a waits-for graph (WFG), where deadlocks appear as cycles.
- To resolve a deadlock, a *victim* transaction is selected and is forced to abort.

![Diagram showing two transactions, T1 and T2, with a deadlock cycle L1: w1[x] w1[y] wl2[y] w2[y] ...?]
Distributed DBMS Model

transactions

network

data manager

transaction manager

physical database
Serialization of Distributed Logs

- Two operations $P_j(A_x)$ and $Q_i(B_Y)$ conflict if all of the following apply:
  - $P$ and $Q$ are not both READ (*concurrent READs never conflict*)
  - $A = B$ (*both access the same record*)
  - $i \neq j$ (*they belong to different transactions*)
  - $X = Y$ (*both appear in the same log*)

**Theorem:** Distributed logs are serializable if there exists a total ordering of the transactions such that for conflicting operations $P_j$ and $Q_i$ a log shows $P_j \rightarrow Q_i$ only if $T_j \rightarrow T_i$
Distributed Transaction Processing

Transactions:

T₁: WRITE(V);
T₂: READ(X); WRITE(Z);
T₃: READ(W); WRITE(V); READ(Z);
T₄: READ(V); READ(Z);

Logs:

L₁: R₄(V₁) W₃(V₁) R₂(X₁) W₁(V₁)
L₂: R₃(W₂) W₃(V₂) R₁(W₂) W₁(Z₂) W₁(V₂) W₂(Z₂)
L₃: R₃(Z₃) W₃(V₃) R₁(V₃) R₄(Z₃) W₁(V₃) W₂(Z₃)

Are these logs equivalent to some serial execution of the transactions?
Serialization of Distributed Logs

Conflict: $P_j(A_X)$ and $Q_i(B_Y)$ conflict if

1. $P$ and $Q$ are not both READ, and
2. $A = B$, and
3. $i \neq j$, and
4. $X = Y$

$L_1: R_4(V_1) W_3(V_1) R_2(X_1) W_1(V_1)$

$L_2: R_3(W_2) W_3(V_2) R_1(W_2) W_1(Z_2) W_1(V_2) W_2(Z_2)$

$L_3: R_3(Z_3) W_3(V_3) R_1(V_3) R_4(Z_3) W_1(V_3) W_2(Z_3)$
Serializability Graph

Graph is cycle free → Serializable

- a) $R_4(V_1) → W_3(V_1)$ [T$_4$ → T$_3$]
- b) $R_4(V_1) → W_1(V_1)$ [T$_4$ → T$_1$]
- c) $W_3(V_1) → W_1(V_1)$ [T$_3$ → T$_1$]
- d) $W_3(V_2) → W_1(V_2)$ [T$_3$ → T$_1$]
- e) $W_1(Z_2) → W_2(Z_2)$ [T$_1$ → T$_2$]
- f) $R_3(Z_3) → W_2(Z_3)$ [T$_3$ → T$_2$]
- g) $W_3(V_3) → R_1(V_3)$ [T$_3$ → T$_1$]
- h) $W_3(V_3) → W_1(V_3)$ [T$_3$ → T$_1$]
- i) $R_4(Z_3) → W_2(Z_3)$ [T$_4$ → T$_2$]