Transaction Processing Overview

Transaction processing studied in depth in
CS W4112-Database System Implementation
Transactions

- A user program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read from, and written to, the database.
- A transaction is the DBMS’s abstract view of a user program, as simply a sequence of reads and writes to the database.

A transaction is a series of actions (Reads and Writes) on a database that form a “logical unit.”

Example: all database actions required to transfer money from one bank account to another.

**ACID properties** of transactions:

- **Atomicity** (enforced via log + recovery protocol)
- **Consistency** (enforced by DBMS)
- **Isolation** (enforced by concurrency control protocol)
- **Durability** (enforced via log + recovery protocol)

Transactions either **commit** (when they complete successfully) or **abort** (when they don’t).
Atomicity of Transactions

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are atomic, so we can think of a transaction as always either:
  - Executing all its actions in one step, or
  - Not executing any actions at all
- The DBMS logs all actions so that it can undo the actions of aborted transactions.

Consistency of Transactions

- The DBMS doesn’t understand the semantics of the data beyond the constraints that are defined as part of the database schema.
- The DBMS makes sure all such constraints hold.
- A transaction can cause temporary violations of database constraints while it is still in progress, but everything should be back to normal when the transaction commits.
- Specifically, if the database is in a consistent state (i.e., all constraints hold) when a transaction starts executing, then the database is back in a consistent state after the transaction commits.
Isolation of Transactions

- Concurrent execution of transactions is essential for good DBMS performance
- Concurrency is achieved by the DBMS, which interleaves actions (reads and writes of database objects) of various transactions
- Users submit transactions, and can think of each transaction as executing by itself, logically speaking
- The DBMS supports this “illusion” via a concurrency control protocol such as Strict 2-Phase Locking

Durability of Transactions

- If a transaction commits (i.e., it completes successfully), then its effects on the database have to persist forever and cannot be forgotten
- Any changes to the database have to survive system crashes and even natural disasters
- The DBMS logs all actions so that it can redo the actions of committed transactions if needed, to guarantee transaction durability
Concurrency Control, for Isolation

Consider two transactions T1 and T2:

- T1 transfers $100 from account B to account A
- T2 gives each account 2% interest

If T1 and T2 are submitted to the database at about the same time, there is no guarantee that T1 will execute before T2 or vice versa. However, the net effect must be equivalent to T1 and T2 running serially in some order (i.e., T1 followed by T2, or T2 followed by T1).

One possible interleaving (or “schedule”) of the actions of T1 and T2

- T1: $A = A + 100$, $B = B - 100$
- T2: $A = 1.02A$, $B = 1.02B$

The schedule is OK, because it’s logically equivalent—in terms of its effects on the database contents—to executing T1 fully, followed by executing T2 fully (i.e., serial schedule T1; T2):

- T1: $A = A + 100$, $B = B - 100$
- T2: $A = 1.02A$, $B = 1.02B$

The two schedules above are equivalent because we can get from one to the other by swapping the order of $B = B - 100$ and $A = 1.02A$, which are nonconflicting actions affecting different objects in the database (i.e., B and A, respectively).
Another possible interleaving (or “schedule”) of the actions of T1 and T2

\[
\begin{align*}
T1: & \quad A &= A + 100, \quad B = B - 100 \\
T2: & \quad A &= 1.02 \times A, \quad B = 1.02 \times B
\end{align*}
\]

The schedule is not OK, because it’s not logically equivalent—in terms of its effects on the database contents—to either T1; T2 or T2; T1

Concurrency Control: Some Definitions

- **Serial schedule** is a schedule that does not interleave the actions of its transactions (i.e., each transaction in the schedule is fully executed before the next transaction is fully executed, and so on)
- Two schedules are **equivalent** if, for any database state, the effect on the database of executing the first schedule is identical to the effect of executing the second schedule
- **Serializable schedule** is a schedule that is equivalent to a serial schedule of the committed transactions
Transaction Schedules as Series of Reads and Writes

From now on, we will view transactions and schedules as sequences of reads and writes, without worrying about the actual values that are read or written.

So we will write a schedule such as:

\[
\begin{align*}
\text{T1:} & \quad A=A+100, \quad B=B-100 \\
\text{T2:} & \quad A=1.02*A, \quad B=1.02*B
\end{align*}
\]

simply as reads R and writes W, where the number next to R or W identifies the corresponding transaction (e.g., R1(A) is a read of object A by transaction T1)

\[
\begin{align*}
\text{T1:} & \quad R1(A) \ W1(A) \quad R1(B) \ W1(B) \\
\text{T2:} & \quad R2(A) \ W2(A) \quad R2(B) \ W2(B)
\end{align*}
\]

or sometimes, equivalently, in one line as:

\[
R1(A) \ W1(A) \ R2(A) \ W2(A) \ R1(B) \ W1(B) \ R2(B) \ W2(B)
\]

Anomalies with Interleaved Executions of Transactions

- **Reading Uncommitted Data ("dirty reads"):**

\[
\begin{align*}
\text{T1:} & \quad R1(A) \ W1(A) \quad R1(B) \ W1(B) \text{Abort1} \\
\text{T2:} & \quad R2(A) \ W2(A) \text{Commit2}
\end{align*}
\]

- **Unrepeatable Reads:**

\[
\begin{align*}
\text{T1:} & \quad R1(A) \quad R1(A) \ W1(A) \text{Commit1} \\
\text{T2:} & \quad R2(A) \ W2(A) \text{Commit2}
\end{align*}
\]
Anomalies (Continued)

• Overwriting Uncommitted Data:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Operations</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>W1(A) W1(B)</td>
<td>Commit1</td>
</tr>
<tr>
<td>T2:</td>
<td></td>
<td>Commit2</td>
</tr>
</tbody>
</table>

All of these schedules are problematic (they are not serializable)

How can we avoid them?

Strict 2-Phase Locking (Strict 2PL) Protocol

Strict 2PL allows only serializable schedules

• **Before reading** an object A, a transaction must have requested and hold a **shared lock** on A (i.e., S(A))
• Multiple transactions can hold a shared lock on an object simultaneously
• **Before writing** an object A, a transaction must have requested and hold an **exclusive lock** on A (i.e., X(A))
• At most one transaction can hold an exclusive lock on an object and no transactions can hold a shared lock on the object
• Transactions only **release their locks (U) when they complete** (i.e., as part of COMMIT or ABORT)
What are the 2 phases in Strict 2PL?

- Phase 1: execution of the transaction, where locks are acquired but never released
- Phase 2: COMMIT or ABORT of the transaction, where all locks are finally released

Examples of schedules possible under Strict 2PL? On board…