CHAPTER: TRANSACTIONS
CHAPTER 14: TRANSACTIONS

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.
**Transaction Concept**

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

- E.g. transaction to transfer $50 from account A to account B:
  1. **read**($A$)
  2. $A := A - 50$
  3. **write**($A$)
  4. **read**($B$)
  5. $B := B + 50$
  6. **write**($B$)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions
EXAMPLE OF FUND TRANSFER

- Transaction to transfer $50 from account A to account B:
  1. read(A)
  2. A := A – 50
  3. write(A)
  4. read(B)
  5. B := B + 50
  6. write(B)

- Atomicity requirement
  - if the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
Example of Fund Transfer (Cont.)

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
Example of Fund Transfer (Cont.)

- Transaction to transfer $50 from account A to account B:
  1. read\((A)\)
  2. \(A := A - 50\)
  3. write\((A)\)
  4. read\((B)\)
  5. \(B := B + 50\)
  6. write\((B)\)

- **Consistency requirement** in above example:
  - the sum of A and B is unchanged by the execution of the transaction
Example of Fund Transfer (Cont.)

- In general, consistency requirements include:
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency
**Example of Fund Transfer (Cont.)**

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

  T1

  1. read($A$)
  2. $A := A - 50$
  3. write($A$)

  T2

  4. read($B$)
  5. $B := B + 50$
  6. write($B$)

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.

- However, executing multiple transactions concurrently has significant benefits, as we will see later.
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.

- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
**ACID Properties (Cont.)**

- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$, finished execution before $T_i$ started, or $T_j$ started execution after $T_i$ finished.
Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.
Transaction State (Cont.)

- Active
- Partially Committed
- Committed
- Failed
- Aborted
**CONCURRENT EXECUTIONS**

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
CONCURRENT EXECUTIONS

- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- Will study in Chapter 16, after studying notion of correctness of concurrent executions.
**Schedules**

- **Schedule** – a sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
Schedules

- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement
**Schedule 1**

- Let $T_1$ transfer $50$ from $A$ to $B$, and $T_2$ transfer $10\%$ of the balance from $A$ to $B$.
- A [serial schedule](#) in which $T_1$ is followed by $T_2$:

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($A$)</td>
<td>read ($A$)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$temp := A * 0.1$</td>
</tr>
<tr>
<td>write ($A$)</td>
<td>$A := A - temp$</td>
</tr>
<tr>
<td>read ($B$)</td>
<td>write ($A$)</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>read ($B$)</td>
</tr>
<tr>
<td>write ($B$)</td>
<td>$B := B + temp$</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>
**Schedule 2**

- A serial schedule where $T_2$ is followed by $T_1$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $(A)$</td>
<td>read $(B)$</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$B := B + 50$</td>
</tr>
<tr>
<td>write $(A)$</td>
<td>write $(B)$</td>
</tr>
<tr>
<td>$A := A - temp$</td>
<td>$B := B + temp$</td>
</tr>
<tr>
<td>$temp := A * 0.1$</td>
<td></td>
</tr>
</tbody>
</table>

commit

commit
Let $T_1$ and $T_2$ be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to Schedule 1.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (A)</td>
<td>read (A)</td>
</tr>
<tr>
<td>$A := A - 50$</td>
<td>$temp := A * 0.1$</td>
</tr>
<tr>
<td>write (A)</td>
<td>$A := A - temp$</td>
</tr>
<tr>
<td></td>
<td>write (A)</td>
</tr>
<tr>
<td>read (B)</td>
<td>read (B)</td>
</tr>
<tr>
<td>$B := B + 50$</td>
<td>$B := B + temp$</td>
</tr>
<tr>
<td>write (B)</td>
<td>write (B)</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>

In Schedules 1, 2 and 3, the sum $A + B$ is preserved.
**Schedule 4**

- The following concurrent schedule does not preserve the value of \((A + B)\).

<table>
<thead>
<tr>
<th>(T_1)</th>
<th>(T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ((A))</td>
<td>read ((A))</td>
</tr>
<tr>
<td>(A := A - 50)</td>
<td>(temp := A * 0.1)</td>
</tr>
<tr>
<td>write ((A))</td>
<td>(A := A - temp)</td>
</tr>
<tr>
<td>read ((B))</td>
<td>write ((A))</td>
</tr>
<tr>
<td>(B := B + 50)</td>
<td>read ((B))</td>
</tr>
<tr>
<td>write ((B))</td>
<td>write ((B))</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td>(B := B + temp)</td>
</tr>
</tbody>
</table>
Serializability

A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. conflict serializability
2. view serializability
CONFLICTING INSTRUCTIONS (OPERATION)

Instructions $l_i$ and $l_j$ of transactions $T_i$ and $T_j$ respectively, conflict if and only if there exists some item $Q$ accessed by both $l_i$ and $l_j$, and at least one of these instructions wrote $Q$.

1. $l_i = \text{read}(Q)$, $l_j = \text{read}(Q)$. $l_i$ and $l_j$ don’t conflict.

2. $l_i = \text{read}(Q)$, $l_j = \text{write}(Q)$. They conflict.

3. $l_i = \text{write}(Q)$, $l_j = \text{read}(Q)$. They conflict.

4. $l_i = \text{write}(Q)$, $l_j = \text{write}(Q)$. They conflict.
LOST UPDATE PROBLEM (W-W CONFLICT)

- T1
  - R(X)
  - X = X - 10
  - W(X)
  - R(Y)
  - Y = Y + 10
  - W(Y)
  - Commit

- T2
  - R(X)
  - X = X + 20
  - W(X)
  - Commit
**Dirty Read Problem (W-R Conflict)**

- **T1**
  - R(X)
  - X = X - 10
  - W(X)

- **T2**
  - R(X)
  - X = X + 10
  - W(X)
  - Commit

- ROLLBACK
Unrepeated Read Problem (R-W Conflict)

- T1
  - R(X)

- T2
  - R(X)
  - X = X + 10
  - W(X)

- R(X)
We say that a schedule $S$ is conflict serializable if it is conflict equivalent to a serial schedule.

If a schedule $S$ can be transformed into a schedule $S'$ by a series of swaps of non-conflicting instructions, we say that $S$ and $S'$ are conflict equivalent.
Schedule 3 can be transformed into Schedule 6, a serial schedule where $T_2$ follows $T_1$, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (A)</td>
<td>read (A)</td>
</tr>
<tr>
<td>write (A)</td>
<td>write (A)</td>
</tr>
<tr>
<td>read (B)</td>
<td>read (B)</td>
</tr>
<tr>
<td>write (B)</td>
<td>write (B)</td>
</tr>
</tbody>
</table>

Schedule 3

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (A)</td>
<td>read (A)</td>
</tr>
<tr>
<td>write (A)</td>
<td>write (A)</td>
</tr>
<tr>
<td>read (B)</td>
<td>read (B)</td>
</tr>
<tr>
<td>write (B)</td>
<td>write (B)</td>
</tr>
</tbody>
</table>

Schedule 6
Example of a schedule that is not conflict serializable:

<table>
<thead>
<tr>
<th></th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($Q$)</td>
<td></td>
<td>write ($Q$)</td>
</tr>
<tr>
<td>write ($Q$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We are unable to swap instructions in the above schedule to obtain either the serial schedule $< T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$. 

**Conflict Serializability (Cont.)**
**View Serializability**

- Let $S$ and $S'$ be two schedules with the same set of transactions. $S$ and $S'$ are **view equivalent** if the following three conditions are met, for each data item $Q$,
  
  1. If in schedule $S$, transaction $T_i$ reads the initial value of $Q$, then in schedule $S'$ also transaction $T_i$ must read the initial value of $Q$.
  
  2. If in schedule $S$ transaction $T_i$ executes $\text{read}(Q)$, and that value was produced by transaction $T_j$ (if any), then in schedule $S'$ also transaction $T_i$ must read the value of $Q$ that was produced by the same $\text{write}(Q)$ operation of transaction $T_j$.
  
  3. The transaction (if any) that performs the final $\text{write}(Q)$ operation in schedule $S$ must also perform the final $\text{write}(Q)$ operation in schedule $S'$.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.
**View Serializability (Cont.)**

- A schedule $S$ is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

<table>
<thead>
<tr>
<th>$T_{27}$</th>
<th>$T_{28}$</th>
<th>$T_{29}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $(Q)$</td>
<td>write $(Q)$</td>
<td>write $(Q)$</td>
</tr>
<tr>
<td>write $(Q)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.
OTHER NOTIONS OF SERIALIZABILITY

- The schedule below produces same outcome as the serial schedule \(< T_1, T_5 >\), yet is not conflict equivalent or view equivalent to it.

<table>
<thead>
<tr>
<th></th>
<th>( T_1 )</th>
<th>( T_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (A)</td>
<td>A := A - 50</td>
<td>read (B)</td>
</tr>
<tr>
<td>write (A)</td>
<td></td>
<td>write (B)</td>
</tr>
<tr>
<td>read (B)</td>
<td>B := B + 50</td>
<td></td>
</tr>
<tr>
<td>write (B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Determining such equivalence requires analysis of operations other than read and write.
TESTING FOR SERIALIZABILITY

- Consider some schedule of a set of transactions $T_1, T_2, \ldots, T_n$
- **Precedence graph** — a directed graph where the vertices are the transactions (names).
- We draw an arc from $T_i$ to $T_j$ if the two transaction conflict, and $T_i$ accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**
**Test for Conflict Serializability**

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order $n^2$ time, where $n$ is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where $e$ is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
  - Are there others?
Recoverable Schedules

- **Recoverable schedule** — if a transaction $T_j$ reads a data item previously written by a transaction $T_i$, then the commit operation of $T_i$ appears before the commit operation of $T_j$.

- The following schedule (Schedule 11) is not recoverable if $T_9$ commits immediately after the read

<table>
<thead>
<tr>
<th></th>
<th>$T_8$</th>
<th>$T_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>(A)</td>
<td>read (A)</td>
</tr>
<tr>
<td>write</td>
<td>(A)</td>
<td>commit</td>
</tr>
<tr>
<td>read</td>
<td>(B)</td>
<td></td>
</tr>
</tbody>
</table>

- If $T_8$ should abort, $T_9$ would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.
**Cascading Rollbacks**

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

<table>
<thead>
<tr>
<th></th>
<th>$T_{10}$</th>
<th>$T_{11}$</th>
<th>$T_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read (A)</td>
<td>read (A)</td>
<td>read (A)</td>
</tr>
<tr>
<td></td>
<td>read (B)</td>
<td>write (A)</td>
<td>write (A)</td>
</tr>
<tr>
<td></td>
<td>write (A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If $T_{10}$ fails, $T_{11}$ and $T_{12}$ must also be rolled back.

- Can lead to the undoing of a significant amount of work