Transactions

Chapter 13: Transactions

• Testing for Serializability
• Transaction Definition in SQL
• Implementation of Isolation
• Recoverability
• Serializability
• Concurrent Executions
• Implementation of Atomicity and Durability
• Transaction State
• Transaction Concept
consistent.

When the transaction is committed, the database must be consistent.

During transaction execution the database may be inconsistent.

A transaction must see a consistent database.

Possibly updates various data items.

A transaction is a unit of program execution that accesses and
Persist after the transaction successfully completes.

**Durability.** The values changed by the transaction must concurrently executed transactions.

The consistency of the database.

**Isolation.** Each transaction executes as if it is the only one executing in the database, and the database system guarantees that intermediate transaction results are hidden from other transactions that are executing in the system.

**Consistency.** Execution of a transaction in isolation preserves the consistency of the database or none are properly reflected in the database.

**Atomicity.** Either all operations of the transaction are completed or none are completed.

To preserve integrity of data, the database system must ensure the following properties.

**ACID Properties**
execution \( A = 950 \) and \( B = 2050 \)

If initial values are \( A = 1000, B = 2000 \), then after the •

By the execution of the transaction, Consistency requirement — the sum of \( A \) and \( B \) is unchanged •

\[
\begin{align*}
\text{read}(B) & \quad \text{(3: write)(A)} \\
2. \ A & = A - 50 \\
\text{read}(A) & \quad \text{(4: read)(B)} \\
3. \ B & = B + 50 \\
\text{write}(B) & \quad \text{(5: write)(B)} \\
6. \ \text{write} & \end{align*}
\]

Transaction structure •

Transfer $50 from account \( A \) to account \( B \)

Example of Fund Transfer
In the database, or none are.

If atomicity is met, all actions of the transaction are reflected

is in an inconsistent state.

If the system crashes between steps 3 and 6, then the database

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>Last Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020</td>
<td>090</td>
<td>6.</td>
</tr>
<tr>
<td>2000</td>
<td>090</td>
<td>4.</td>
</tr>
<tr>
<td>2000</td>
<td>1000</td>
<td>2.</td>
</tr>
</tbody>
</table>

Example values of database variables at various points in execution:

Example of Fund Transfer (Cont.)
Transaction State

- Aborted, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options:
  - Kill the transaction
  - Restart the transaction - only if no internal logical error.
- Partially committed, after the final statement has been executed.
- Committed, after successful completion.
- Active, the initial state.
• Does not allow transactions to execute concurrently.
  • Copying the entire database.
  • Extremely inefficient: executing a single transaction requires
    implementing in much the same manner as a text-editing
  • (shadow copies) of the database.
  • The shadow-database scheme assumes that only one
    transaction is active at a time, and is based on making copies
    that only one
  • Implements the support for atomicity and durability.
  • The recovery-management component of a database system

**Implementation of Atomicity and Durability**
Each individual transaction must preserve the order in which the instructions appear in instructions of those transactions. A schedule for a set of transactions must consist of all orders in which instructions are executed.

- Schedules – execution sequences that indicate the chronological order from which they destroy the consistency of the database.
<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B ) = ( B ) + ( \text{temp} )</td>
<td>( B ) = ( B ) + ( \text{temp} )</td>
</tr>
<tr>
<td>( \text{read} ) (( B ))</td>
<td>( \text{read} ) (( B ))</td>
</tr>
<tr>
<td>( \text{write} ) (( A ))</td>
<td>( \text{write} ) (( A ))</td>
</tr>
<tr>
<td>( \text{temp} ) - ( A ) = ( A )</td>
<td>( \text{temp} ) - ( A ) = ( A )</td>
</tr>
<tr>
<td>( A \ast 0.1 )</td>
<td>( A \ast 0.5 )</td>
</tr>
</tbody>
</table>

In Schedule 1, balance from \( A \) to \( B \). The following is a serial schedule.

Let Schedule 1 transfer \$500 from \( A \) to \( B \), and Schedule 2 transfer 10% of the

Example Schedules
In both Schedules 1 and 2, the sum $A + B$ is preserved.

<table>
<thead>
<tr>
<th>(write)</th>
<th>(read)</th>
<th>(write)</th>
<th>(read)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B + B = B$</td>
<td>$B$</td>
<td>$B + B - A = A$</td>
<td>$A$</td>
</tr>
<tr>
<td>$B$</td>
<td>$B$</td>
<td>$B - A = A$</td>
<td>$A$</td>
</tr>
<tr>
<td>$A * 0.1'$</td>
<td>$A$</td>
<td>$A$</td>
<td>$A$</td>
</tr>
</tbody>
</table>

Following serial schedule (Schedule 2), $L_2$ is followed by $L_1$. Let $L_1$ and $L_2$ be the transactions defined previously. In the example schedules (cont.)
Example Concurrent Schedule
equivalent to a serial schedule.

A (possibly concurrent) schedule is serializable if it is

The sequence of steps in an execution is called a schedule

Interleaved

In a concurrent execution, steps of a set of transactions may be

Consistency

Serial execution of a set of transactions preserves database

Consistency

Basic Assumption – Each transaction preserves database

Serializability
and write instructions accessing $O$.

In the system, we deal only with read transactions $r_i$ and $r_j$. We shall consider all the cases assuming there were only two

equivalent to a serial schedule.

We say that a schedule $S$ is conflict serializable if it is conflict

are conflict equivalent.

If a schedule $S$ can be transformed into a schedule $S'$ by a

temporal order between them.

Intuitively, a conflict between two instructions forces a (logical)

both $r_i$ and $r_j$ and at least one of these instructions wrote

conflict if and only if there exists some item $O$ accessed by

instructions $I_i$ and $I_j$ of transactions $r_i$ and $r_j$, respectively.
In this case `L` came (logically) after `L`, and `L` created the value of the same value as both `L` and `L` (at least on the basis of the listed accesses) as both `L` and `L`.

### Conflicts Between Two Transactions

<table>
<thead>
<tr>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L</code></td>
<td><code>L</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Read</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L</code></td>
<td><code>L</code></td>
</tr>
</tbody>
</table>
In this case \( \text{\$L} \) came (logically after \( \text{\$L} \)) as \( \text{\$L} \) overwrote the value created by \( \text{\$L} \).

<table>
<thead>
<tr>
<th>Write</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{$L} )</td>
<td>( \text{$L} )</td>
</tr>
</tbody>
</table>

In this case \( \text{\$L} \) came (logically after \( \text{\$L} \)) as \( \text{\$L} \) read the value created by \( \text{\$L} \).

<table>
<thead>
<tr>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{$L} )</td>
<td>( \text{$L} )</td>
</tr>
</tbody>
</table>

Conflicts Between Two Transactions (Cont.)
3. For each data item \( \emptyset \), the transaction (if any) that performs the final write operation in schedule \( S \) must perform the final write operation in schedule \( S' \).

2. For each data item \( \emptyset \), if transaction \( \emptyset \) executes read \( \emptyset \) in schedule \( S \) and that value was produced by transaction \( L \), then transaction \( L \) must in schedule \( S' \) also read the value of \( \emptyset \) of that was produced by transaction \( L \).

1. For each data item \( \emptyset \), if transaction \( \emptyset \) reads the initial value of \( \emptyset \) in schedule \( S \), then transaction \( S' \) must in schedule \( S' \) also read the initial value of \( \emptyset \).

Let \( S \) and \( S' \) be two schedules with the same set of transactions. 

View Serializability
<table>
<thead>
<tr>
<th>( \text{write} )</th>
<th>( \text{write} )</th>
<th>( \text{write} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L )</td>
<td>( L )</td>
<td>( L )</td>
</tr>
</tbody>
</table>

Schedule 9 (from text) — a view-serializable schedule.

- There are view serializable schedules that are not conflict serializable.
- Every conflict serializable schedule is view serializable, but a schedule \( S \) is view serializable if it is view equivalent to a

**View Serializability (cont.)**
appears before the read operation of \( T \),

items previously written by \( T \), the commit operation of \( T \),

\( T \) and \( T \) such that \( T \) reads a data

each pair of transactions \( T \) and \( T \) cannot occur,

Cascadable schedules — cascading rollbacks do not have the potential of cascading rollback.

— Useful to design protocols for transaction processing that do

— Undesirable since it leads to the undoing of a significant

series of transaction rollbacks.

Cascading rollback — a single transaction failure leads to a

"Recoverability"
Concurrency control schemes traded off the amount of concurrency they allow and the amount of overhead that they incur.

- Concurrency control schemes traded off the amount of concurrency.
- A locking policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of cascadability.
- Serial schedules must be conflict or view serializable and for database consistency and safe handling of transactions.

Implementation of Isolation
- Read uncommitted
- Read committed
- Repeatable read
- Serializable — default

Levels of consistency specified by SQL-92:

- **Rollback work** causes current transaction to abort.
- New one.
- **Commit work** commits current transaction and begins a new transaction in SQL ends by:

  - In SQL, a transaction begins implicitly.
  - Specifying the set of actions that comprise a transaction.

Data manipulation language must include a construct for

**Transaction Definition in SQL**
Example 1:

We may label the arc by the item that was accessed.

We draw an arc from $L_j^i$ to $L_j^i$ if the two transactions conflict.

And $L_j^i$ made the relevant access earlier.

We draw an arc from $L_j^i$ to $L_j^i$ (names).

A directed graph — a precedence graph — where the vertices are the transactions.

Consider some schedule of a set of transactions $L_j^i$.

Testing for Serializability
and is therefore equivalent to some serial schedule.
2. In Example 2, the schedule had a graph which is acyclic,
therefore is not equivalent to any serial schedule.
1. Example 1's schedule had a graph containing a cycle and

- Example: •
  
  acyclic.
  embedded in a linear graph. That is, the precedence graph is
  •
  A schedule is serializable if its precedence graph can be

Test for Conflict Serializability
avoid nonserializable schedules. Being created; instead a protocol will impose the discipline that be generated not examine the precedence graph as it is generally not develop protocols that will assure serializability.

Goal: to develop protocols that will assure serializability.

- Falls in the class of \( NP \)-complete problems.
- Exhaustive testing of all possible distinct graphs, this problem.
- The algorithm for determining view serializability requires expensive in CPU time.
- Possible distinct labeled graphs for a graph without a cycle;
- Construct a labeled precedence graph and search over all;
- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.

Test for View Serializability