Transactions

• A transaction is specified by a client as a sequence of operations on objects to be performed as an indivisible unit by the servers managing those objects.
• Goal is to ensure that all objects managed by a server remain in a consistent state
  – when they are accessed by multiple transactions
  – when the server crashes

A Banking Example

• Three accounts: A-$100, B-$200, C-$300
  – Client 1: transfer $4 from A to B
  – Client 2: transfer $3 from C to B
• Concurrent execution of transactions can produce inconsistent result unless certain properties are imposed on the accesses
ACID Properties of Transactions

- **Atomicity**: A transaction either completes successfully or has no effect at all (i.e., all or nothing)
- **Consistency**: A transaction takes the system from one consistent state to another consistent state
- **Isolation**: Each transaction must be performed without interference from other transactions (i.e., the intermediate effects of a transaction must not be visible to other transactions)
- **Durability**: Once a transaction commits, the results are saved in permanent storage. No failure after a commit can undo the results or cause them to get lost.

<table>
<thead>
<tr>
<th></th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A: $100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write A: $96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read B: $200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write B: $204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read C: $300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write C: $297</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read B: $204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write B: $207</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Isolation is achieved when transactions are performed serially.

Transaction Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END.Transactional</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, etc.</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, etc.</td>
</tr>
</tbody>
</table>

An airline reservation example:

**BEGIN_TRANSACTION**
- Reserve a seat for NY to Paris
- Reserve a seat for Paris to Athens
- Reserve a seat for Athens to Hong Kong

**END_TRANSACTION**
Distributed Transactions

• A **distributed transaction** is a transaction that accesses objects managed by multiple servers
  – All servers involved must agree whether to commit or abort
• A distributed transaction can be structured in two ways:
  – In a **flat transaction**, operations are invoked sequentially
  – In a **nested transaction**, the top-level transaction can start subtransactions, and each subtransaction can start further subtransactions down to any depth of nesting

BEGIN_TRANSACTION
reserv a seat for DSM → LAX;
reserve a hotel room;
reserve a rental car;
END_TRANSACTION

BEGIN_TRANSACTION
BEGIN_SUBTRANSACTION
reserve a seat for DSM → LAX;
END_SUBTRANSACTION
BEGIN_SUBTRANSACTION
reserve a hotel room;
END_SUBTRANSACTION
BEGIN_SUBTRANSACTION
reserve a rental car;
END_SUBTRANSACTION
END_TRANSACTION

A flat transaction

A nested transaction
Advantages of Nested Transactions

• Subtransactions at the same level can run concurrently
  – When subtransactions run in different servers, they can work in parallel
• Subtransactions can commit or abort independently
  – When a subtransaction aborts, the parent can decide whether to abort or not
  – With a flat transaction, one transaction failure would cause the whole transaction to be aborted

Achieving Atomicity: Private Workspace

• When a process starts a transaction, it’s given a private workspace containing all the objects to which it has access
  – Changes are made to the private workspace
• On Commit: copy private workspace to stable storage
• On Abort: just delete private workspace
• Optimization: Don’t make copies for reads
Private Workspace Example

(a) The file index and disk blocks for a three-block file
(b) The situation after a transaction has modified block 0 and appended block 3
(c) After committing

Achieving Atomicity: Write-Ahead Logs

- Objects are modified in place
- Prior to making a change, a record is written to the write-ahead log in stable storage
  - A log record is a 4-tuple <transaction ID, object ID, old value, new value>
- If the transaction commits, a commit record is written to the log.
- If the transaction aborts, the log is used to back up to the original state (this is called a rollback)
Write-Ahead Log Example

\[
\begin{array}{ccc}
\text{x = 0;} & \text{Log} & \text{Log} \\
\text{y = 0;} & \text{Log} & \text{Log} \\
\text{BEGIN TRANSACTION;} & [x = 0 / 1] & [x = 0 / 1] \\
\text{x = x + 1;} & [y = 0 / 2] & [y = 0 / 2] \\
\text{y = y + 2} & [x = 1 / 4] \\
\text{x = y * y;} & \text{(a)} & \text{(b)} & \text{(c)} & \text{(d)} \\
\text{END TRANSACTION;} & & & & \\
\end{array}
\]

a) A transaction. b)-d) The log before each statement is executed.

Concurrent Control

- **Concurrency control** is used to achieve isolation (i.e., the intermediate effects of a transaction must not be visible to other transactions)
- The goal of concurrency control is to control the execution of concurrent transactions such that the result is the same as if they executed in some sequential order
- A **schedule** of transactions is an interleaving of the operations of concurrent transactions
- A **legal schedule** (or a **serially equivalent schedule**) is one that provides the same result as though the transactions ran sequentially in some order
- A **scheduler** employs a concurrency control algorithm to produce legal schedules
An Example

Conflicting Operations

- **Conflicting operations** are those operations that operate on the same data item and whose combined effects depend on the order in which they are executed.
- Two operations $O_i(x)$ and $O_j(x)$ of transactions $T_i$ and $T_j$ conflict if at least one of the operations is a write
  - read-write conflict, write-write conflict
- A schedule is serially equivalent if all pairs of conflicting operations of two transactions are executed in the same order.
## An Example

<table>
<thead>
<tr>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=read(i);</td>
<td>write(i, 55);</td>
</tr>
<tr>
<td>write(j, 44);</td>
<td>write(j, 66);</td>
</tr>
</tbody>
</table>

A serially equivalent schedule

<table>
<thead>
<tr>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=read(i);</td>
<td>write(i, 55);</td>
</tr>
<tr>
<td>write(j, 44);</td>
<td>write(j, 66);</td>
</tr>
</tbody>
</table>

A non-serially equivalent schedule