Goals: Understand the basic properties of a transaction and learn the concepts underlying transaction processing as well as the concurrent executions of transactions.

A transaction is a unit of a program execution that accesses and possibly modifies various data objects (tuples, relations).

DBMS has to maintain the following properties of transactions:

- Atomicity: A transaction is an atomic unit of processing, and it either has to be performed in its entirety or not at all.

- Consistency: A successful execution of a transaction must take a consistent database state to a (new) consistent database state. (\(\leadsto\) integrity constraints)

- Isolation: A transaction must not make its modifications visible to other transactions until it is committed, i.e., each transaction is unaware of other transactions executing concurrently in the system. (\(\leadsto\) concurrency control)

- Durability: Once a transaction has committed its changes, these changes must never get lost due to subsequent (system) failures. (\(\leadsto\) recovery)
Model used for representing database modifications of a transaction:

- **read**(A,x): assign value of database object A to variable x;
- **write**(x,A): write value of variable x to database object A

**Example** of a Transaction $T$

```plaintext
read(A,x)
x := x - 200
write(x,A)
read(B,y)
y := y + 100
write(y,B)
```

**Main focus here:** Maintaining isolation in the presence of multiple, concurrent user transactions

**Goal:** “Synchronization” of transactions; allowing concurrency (instead of insisting on a strict serial transaction execution, i.e., process complete $T_1$, then $T_2$, then $T_3$ etc.)

\[\leadsto\] increase the throughput of the system,
\[\leadsto\] minimize response time for each transaction

Problems that can occur for certain transaction schedules without appropriate concurrency control mechanisms:
Lost Update

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction $T_1$</th>
<th>Transaction $T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read($A,x$)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x:=x+200$</td>
<td>read($A,y$)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$y:=y+100$</td>
</tr>
<tr>
<td>4</td>
<td>write($x,A$)</td>
<td>write($y,A$)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

The update performed by $T_1$ gets lost; possible solution: $T_1$ locks/unlocks database object $A$  
$\Rightarrow$ $T_2$ cannot read $A$ while $A$ is modified by $T_1$

Dirty Read

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction $T_1$</th>
<th>Transaction $T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read($A,x$)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$x:=x+100$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>write($x,A$)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>read($A,y$)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>write($y,B$)</td>
</tr>
<tr>
<td>6</td>
<td>rollback</td>
<td></td>
</tr>
</tbody>
</table>

$T_1$ modifies db object, and then the transaction $T_1$ fails for some reason. Meanwhile the modified db object, however, has been accessed by another transaction $T_2$. Thus $T_2$ has read data that “never existed”.
### Inconsistent Analysis (Incorrect Summary Problem)

<table>
<thead>
<tr>
<th>Time</th>
<th>Transaction $T_1$</th>
<th>Transaction $T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read($A,y_1$)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>read($A,x_1$)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$x_1 := x_1 - 100$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>write($x_1,A$)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>read($C,x_2$)</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>$x_2 := x_2 + x_1$</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>write($x_2,C$)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>9</td>
<td>read($B,y_2$)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>read($C,y_3$)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>sum := $y_1 + y_2 + y_3$</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

In this schedule, the total computed by $T_1$ is wrong (off by 100).  
$\implies T_1$ must lock/unlock several db objects
**Serializability**

DBMS must control concurrent execution of transactions to ensure read consistency, i.e., to avoid dirty reads etc.

\( \sim \) A (possibly concurrent) schedule \( S \) is **serializable** if it is equivalent to a serial schedule \( S' \), i.e., \( S \) has the same result database state as \( S' \).

How to ensure serializability of concurrent transactions?

Conflicts between operations of two transactions:

\[
\begin{array}{c|c|c|c|c}
T_i & T_j & T_i & T_j \\
\hline
\text{read}(A,x) & \text{read}(A,y) & \text{read}(A,x) & \text{write}(y,A) \\
\end{array}
\]

(order does not matter)

\[
\begin{array}{c|c|c|c|c}
T_i & T_j & T_i & T_j \\
\hline
\text{write}(x,A) & \text{read}(A,y) & \text{write}(x,A) & \text{write}(y,A) \\
\end{array}
\]

(order matters)

A schedule \( S \) is **serializable** with regard to the above conflicts iff \( S \) can be transformed into a serial schedule \( S' \) by a series of swaps of non-conflicting operations.
Checks for serializability are based on *precedence graph* that describes dependencies among concurrent transactions; if the graph has no cycle, then the transactions are serializable.

→ they can be executed concurrently without affecting each others transaction result.

**Concurrency Control: Lock-Based Protocols**

- One way to ensure serializability is to require that accesses to data objects must be done in a mutually exclusive manner.

- Allow transaction to access data object only if it is currently holding a **lock** on that object.

- Serializability can be guaranteed using locks in a certain fashion

⇒ Tests for serializability are redundant !

**Types of locks** that can be used in a transaction T:

- **slock**(X): shared-lock (read-lock); no other transaction than T can write data object X, but they can read X

- **xlock**(X): exclusive-lock; T can read/write data object X; no other transaction can read/write X, and

- **unlock**(X): unlock data object X
**Lock-Compatibility Matrix:**

<table>
<thead>
<tr>
<th>requested lock</th>
<th>existing lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>slock</td>
<td>lock</td>
</tr>
<tr>
<td>slock</td>
<td>OK</td>
</tr>
<tr>
<td>xlock</td>
<td>No</td>
</tr>
</tbody>
</table>

E.g., \textit{xlock}(A) has to wait until all \textit{slock}(A) have been released.

**Using locks** in a transaction (lock requirements, LR):

- before each \texttt{read}(X) there is either a \texttt{xlock}(X) or a \texttt{slock}(X) and no \texttt{unlock}(X) in between
- before each \texttt{write}(X) there is a \texttt{xlock}(X) and no \texttt{unlock}(X) in between
- a \texttt{slock}(X) can be tightened using a \texttt{xlock}(X)
- after a \texttt{xlock}(X) or a \texttt{slock}(X) sometime an \texttt{unlock}(X) must occur

**But:** “Simply setting locks/unlocks is not sufficient”

replace each \texttt{read}(X) $\rightarrow$ \texttt{slock}(X); \texttt{read}(X); \texttt{unlock}(X), and \texttt{write}(X) $\rightarrow$ \texttt{xlock}(X); \texttt{write}(X); \texttt{unlock}(X)
Two-Phase Locking Protocol (TPLP)

A transaction T satisfies the TPLP iff

- after the first unlock(X) no locks xlock(X) or slock(X) occur
- That is, first T obtains locks, but may not release any lock (growing phase)
  and then T may release locks, but may not obtain new locks (shrinking phase)

Strict Two-Phase Locking Protocol:

All unlocks at the end of the transaction T ⇒ no dirty reads are possible, i.e., no other transaction can write the (modified) data objects in case of a rollback of T.

Concurrency Control in PostgreSQL

In PostgreSQL (or Oracle) the user can specify the following locks on relations and tuples using the command

\[
\text{lock table in } \langle\text{mode}\rangle \text{ mode;}
\]

<table>
<thead>
<tr>
<th>mode</th>
<th>tuple level</th>
<th>relation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>row share</td>
<td>slock</td>
<td>intended slock</td>
</tr>
<tr>
<td>row exclusive</td>
<td>xlock</td>
<td>intended xlock</td>
</tr>
<tr>
<td>share</td>
<td>—</td>
<td>slock</td>
</tr>
<tr>
<td>share row exclusive</td>
<td>—</td>
<td>sixlock</td>
</tr>
<tr>
<td>exclusive</td>
<td>—</td>
<td>xlock</td>
</tr>
</tbody>
</table>
The following locks are performed automatically by the scheduler:

- **select** → no lock
- **insert/update/delete** → **xlock** /row exclusive
- **select . . . for update** → **slock** /row share
- **commit** → releases all locks

PostgreSQL (and Oracle) furthermore provide *isolation levels* that can be specified before a transaction by using the command

```
set transaction isolation level <level>;
```

- **read committed** (default): each query executed by a transaction sees the data that was committed before the query (not the transaction!)

  \[ (\sim \text{statement level read consistency}) \]

  \[
  \begin{array}{c|c}
  T_1 & T_2 \\
  \hline
  \text{select A from R} & \text{update R set A = new} \\
  \rightarrow \text{old value} & \\
  \text{select A from R} & \text{commit} \\
  \rightarrow \text{old value} & \\
  \text{select A from R} & \text{new value} \\
  \rightarrow & \\
  \end{array}
  \]

  Non-repeatable reads (same select statement in TA gives different results at different times) possible; dirty-reads are not possible
• **serializable**: serializable TAs see only those changes that were committed at the time the TA began, plus own changes.

PostgreSQL generates an error when such a transaction tries to update or delete data modified by a transaction that commits after the serializable transaction began.

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
\text{set transaction isolation level serializable} & \text{set transaction . . .} \\
\text{update R set A = new where B = 1} & \text{update R set A = new where B = 1} \\
\rightarrow \text{ERROR} & \text{commit}
\end{array}
\]

Dirty-reads and non-repeatable reads are not possible. Furthermore, this mode guarantees serializability (but does not provide much parallelism).