Chapter 12

Introduction to Transaction Processing Concepts and Theory (from E&N and my editing)
Outline

- Introduction to Transaction Processing
- Transaction & System Concept
- Concurrency/Synchronization
- Schedule
- Transaction in SQL
Introduction to Transaction Processing

- Single-User Versus Multiuser Systems
- Transactions, Read and Write Operations, and DBMS Buffers
- Why Concurrency Control Is Needed
- Why Recovery Is Needed
• **Single-User System:** At most one user at a time can use the system.

• **Multiuser System:** Many users can access the system concurrently.

• **Concurrency/Synchronization**
  - **Interleaved processing:** concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing:** processes are concurrently executed in multiple CPUs.
• Interleaved processing versus parallel processing of concurrent transactions.
- **A Transaction**: logical unit of database processing that includes one or more access operations (read - retrieval, write - insert or update, delete).

- **A transaction (set of operations)** may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.

- **Transaction boundaries**: Begin and End transaction.

- **An application program** may contain several transactions separated by the Begin and End transaction boundaries.
Basic operations are **read** and **write**

- Basic unit of data transfer from the disk to the computer main memory is **one block**. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

**read_item**(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.

**write_item**(X): Writes the value of program variable X into the database item named X.
Read_Item (X)

- read_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.
Write_Item (X)

- write_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).
Two sample transactions. (a) Transaction $T_1$. (b) Transaction $T_2$

(a) $T_1$

```
read_item (X);
X:=X-N;
write_item (X);
read_item (Y);
Y:=Y+N;
write_item (Y);
```

(b) $T_2$

```
read_item (X);
X:=X+M;
write_item (X);
```
Synchronization

- Ability of Two or More Serial Processes to Interact During Their Execution to Achieve Common Goal
- Recognition that “Today’s” Applications Require Multiple Interacting Processes
  - Client/Server and Multi-Tiered Architectures
  - Inter-Process Communication via TCP/IP
- Fundamental Concern: Address Concurrency
  - Control Access to Shared Information
  - Historically Supported in Database Systems
  - Currently Available in Many Programming Languages
Threat Syn

- Suppose X and Y are Concurrently Executing in Same Address Space
- What are Possibilities?

1

X

Y

2

3

- What Does Behavior at Left Represent?
- Synchronous Execution!
- X Does First Part of Task
- Y Next Part Depends on X
- X Third Part Depends on Y
- Threads Must Coordinate Execution of Their Effort
- Now, What Does Behavior at Left Represent?
- Asynchronous Execution!
- X Does First Part of Task
- Y Does Second Part Concurrent with X Doing Third Part
- What are Issues?

Will Second Part Still Finish After Third Part?
Will Second Part Now Finish Before Third Part?
What Happens if Variables are Shared?

This is the Database Concern - Concurrent Transactions Against Shared Tables!
Concurrent Problem

- The Lost Update Problem. This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

- The Temporary Update (or Dirty Read) Problem. This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.
The Incorrect Summary Problem.
If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.
## Lost update problem

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read_item($X$); $X := X - N$</td>
<td>read_item($X$); $X := X + M$</td>
</tr>
<tr>
<td>2</td>
<td>write_item($X$);</td>
<td>write_item($X$);</td>
</tr>
<tr>
<td>3</td>
<td>read_item($Y$);</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$Y := Y + N$;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>write_item($Y$);</td>
<td></td>
</tr>
</tbody>
</table>

Item $X$ has an incorrect value because its update by $T_1$ is "lost" (overwritten).
Temporary Update Problem

Transaction $T_1$ fails and must change the value of $X$ back to its old value; meanwhile $T_2$ has read the "temporary" incorrect value of $X$. 

\[
\begin{array}{l}
T_1 \\
\text{read}_\text{item}(X); \\
X := X - N; \\
\text{write}_\text{item}(X); \\
\text{read}_\text{item}(Y); \\
\end{array}
\quad
\begin{array}{l}
T_2 \\
\text{read}_\text{item}(X); \\
X := X + M; \\
\text{write}_\text{item}(X); \\
\end{array}
\]
### Incorrect summary problem

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td></td>
<td>sum := 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read_item($A$);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sum := sum + $A$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>read_item($X$);</td>
<td>read_item($X$);</td>
</tr>
<tr>
<td></td>
<td>$X := X - N$;</td>
<td>sum := sum + $X$;</td>
</tr>
<tr>
<td></td>
<td>write_item($X$);</td>
<td>read_item($Y$);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sum := sum + $Y$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$T_3$ reads $X$ after $N$ is subtracted and reads $Y$ before $N$ is added; a wrong summary is the result (off by $N$).</td>
</tr>
</tbody>
</table>
Transaction can be fail → Recovery is needed

- A computer failure (system crash)
  - A hardware or software error occurs
  - The contents of the computer’s internal memory may be lost.
- A transaction or system error
  - Integer overflow
  - Division by zero.
  - Erroneous parameter values or logical programming error
  - The user may interrupt the transaction during its execution.
• **Local errors or exception conditions** detected by the transaction:
  – Data for the transaction may not be found.
  – A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.
  – A programmed abort in the transaction causes it to fail.

• **Concurrency control enforcement:**
  – The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock
• Disk failure:
  Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

• Physical problems and catastrophes:
  This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.
A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

**Transaction states:**
- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State
Transaction States

BEGIN TRANSACTION

READ, WRITE

END TRANSACTION

ACTIVETABLE

PARTIALLY COMMITTED

COMMITTED

ABORT

FAILEDTABLE

TERMINATED

COMMIT
• **begin_transaction:**
  This marks the beginning of transaction execution.

• **read or write:**
  These specify read or write operations on the database items that are executed as part of a transaction.

• **end_transaction:**
  – This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
  – At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.
• **commit_transaction:**
  This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely *committed* to the database and will not be undone.

• **rollback (or abort):**
  This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be *undone*. 
• **undo:**
  Similar to rollback except that it applies to a single operation rather than to a whole transaction.

• **redo:**
  This specifies that certain transaction operations must be redone to ensure that all the operations of a committed transaction have been applied successfully to the database.
Desirable Properties of Transaction

- **Atomicity**: All or nothing
- **Consistency**: Database state consistent upon entry and exit
- **Isolated**: Executed without interference from other transactions
- **Durable**: Changes to the database are permanent after the transaction completes.
System Log

- Keeps track of all transaction operations that affect values of database items
- Log is kept on disk and periodically backed up to guard against catastrophe:
  - Transaction ID
  - [start, TID]
  - [write_item, TID, X, old_value, new_value]
  - [read_item, TID, X]
  - [commit, TID]
  - [abort, TID]
Schedule

- **Transaction schedule or history**
  When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).

- **A schedule (or history) S of n transactions T1, T2, ..., Tn:**
  It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.
Schedule based on Recoverability

- **Recoverable schedule:**
  - One where no transaction needs to be rolled back.
  - A schedule $S$ is *recoverable* if no transaction $T$ in $S$ commits until all transactions $T'$ that have written an item that $T$ reads have committed.

- **Cascadeless schedule:**
  - One where every transaction reads only the items that are written by committed transactions.
  - **Schedules requiring cascaded rollback:** A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
● **Strict Schedules:** A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.
One where no transaction needs to be rolled back. Is guaranteed if

No transaction T in S commits until all transactions T’ that have written an item that T reads has committed

If T2 aborts here then T1 would have to be aborted after commit violating Durability of ACID
Cascadeless Schedule

Those where every transaction reads only the items that are written by committed transactions.

Cascaded Rollback Schedules

A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

<table>
<thead>
<tr>
<th>Start T1</th>
<th>Start T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start T2</td>
<td>Start T2</td>
</tr>
<tr>
<td>R(x) T1</td>
<td>R(x) T1</td>
</tr>
<tr>
<td>W(x) T1</td>
<td>W(x) T1</td>
</tr>
<tr>
<td>R(x) T2</td>
<td>R(y) T1</td>
</tr>
<tr>
<td>R(y) T1</td>
<td>W(y) T1</td>
</tr>
<tr>
<td>W(x) T2</td>
<td>Commit T1</td>
</tr>
<tr>
<td>W(y) T1</td>
<td>R(x) T2</td>
</tr>
</tbody>
</table>

If T1 were to abort here then T2 would have to abort in a cascading fashion. This is a cascaded rollback schedule.
Strict Schedule

A transaction can neither read nor write an item X until the last transaction that wrote X has committed.

(start x = 9)
Start T1
Start T2
R(y) T2
R(x) T1
W(x) T1 (say x = 5)
R(y) T1
W(y) T1
W(x) T2 (say x = 8)

For this example
Say T1 aborts here
Then the recovery process will restore the value of x to 9
Disregarding (x= 8). Although this is cascadeless it is not Strict and the problem needs to be resolved: use REDO
Equivalent Schedule

- Two Schedules $S_1$ and $S_2$ are Equivalent, Denoted As $S_1 \approx S_2$, If and Only If $S_1$ and $S_2$
  - Execute the Same Set of Transactions
  - Produce the Same Results (i.e., Both Take the DB to the Same Final State)
Are the Two Schedules below Equivalent?

$S_1$ and $S_4$ are Equivalent, since They have the Same Set of Transactions and Produce the Same Results

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(X); X:=X− 2; Write(X);</td>
<td>Read(X); X:=X− 2; Write(X);</td>
</tr>
<tr>
<td></td>
<td>Read(Y); Y = Y + 20; Write(Y);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>commit;</td>
<td>commit;</td>
</tr>
</tbody>
</table>

$S_1$: $R_1(X), W_1(X), R_1(Y), W_1(Y), c_1, R_2(X), W_2(X), c_2$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(X); X:=X− 1; Write(X);</td>
<td>Read(X); Y = Y + 20; Write(Y);</td>
</tr>
<tr>
<td></td>
<td>commit;</td>
<td>commit;</td>
</tr>
</tbody>
</table>

$S_4$: $R_1(X), W_1(X), R_2(X), W_2(X), c_2, R_1(Y), W_1(Y), c_1$
Schedule based on Serialization

- **Serial schedule:**
  A schedule $S$ is serial if, for every transaction $T$ participating in the schedule, all the operations of $T$ are executed consecutively in the schedule. Otherwise, the schedule is called nonserial schedule.

- **Serializable schedule:**
  A schedule $S$ is serializable if it is equivalent to some serial schedule of the same $n$ transactions.
Consider $S_1$ and $S_2$ for Transactions $T_1$ and $T_2$

If $X = 10$ and $Y = 20$

- After $S_1$ or $S_2$ $X = 7$ and $Y = 40$

Is $S_3$ a Serializable Schedule?

<table>
<thead>
<tr>
<th>Schedule $S_1$</th>
<th>Schedule $S_2$</th>
<th>Schedule $S_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>Read($X$);</td>
<td>Read($X$);</td>
<td>Read($X$);</td>
</tr>
<tr>
<td>$X := X - 2$;</td>
<td>$X := X - 2$;</td>
<td>$X := X - 2$;</td>
</tr>
<tr>
<td>Write($X$);</td>
<td>Write($X$);</td>
<td>Write($X$);</td>
</tr>
<tr>
<td>Read($Y$);</td>
<td>Read($Y$);</td>
<td>Read($Y$);</td>
</tr>
<tr>
<td>$Y = Y + 20$;</td>
<td>$Y = Y + 20$;</td>
<td>$Y = Y + 20$;</td>
</tr>
<tr>
<td>Write($Y$);</td>
<td>Write($Y$);</td>
<td>Write($Y$);</td>
</tr>
<tr>
<td>commit;</td>
<td>commit;</td>
<td>commit;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_2$</th>
<th>$T_2$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read($X$);</td>
<td>Write($X$);</td>
<td>Read($X$);</td>
</tr>
<tr>
<td>$X := X - 1$;</td>
<td>$X := X - 1$;</td>
<td>$X := X - 1$;</td>
</tr>
<tr>
<td>Write($X$);</td>
<td>Write($X$);</td>
<td>Write($X$);</td>
</tr>
<tr>
<td>Read($Y$);</td>
<td>Read($Y$);</td>
<td>Read($Y$);</td>
</tr>
<tr>
<td>$Y = Y + 20$;</td>
<td>$Y = Y + 20$;</td>
<td>$Y = Y + 20$;</td>
</tr>
<tr>
<td>Write($Y$);</td>
<td>Write($Y$);</td>
<td>Write($Y$);</td>
</tr>
<tr>
<td>commit;</td>
<td>commit;</td>
<td>commit;</td>
</tr>
</tbody>
</table>
Consider $S_1$ and $S_2$ for Transactions $T_1$ and $T_2$

If $X = 10$ and $Y = 20$

- After $S_1$ or $S_2$ $X = 7$ and $Y = 40$

Is $S_4$ a Serializable Schedule?

<table>
<thead>
<tr>
<th>Schedule $S_1$</th>
<th>Schedule $S_2$</th>
<th>Schedule $S_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>Read(X);</td>
<td>Read(X);</td>
<td>Read(X);</td>
</tr>
<tr>
<td>$X:=X-2$;</td>
<td>$X:=X-1$;</td>
<td>$X:=X-2$;</td>
</tr>
<tr>
<td>Write(X);</td>
<td>Write(X);</td>
<td>Write(X);</td>
</tr>
<tr>
<td>Read(Y);</td>
<td>Read(Y);</td>
<td>Read(Y);</td>
</tr>
<tr>
<td>$Y=Y+20$;</td>
<td>$Y=Y+20$;</td>
<td>$Y=Y+20$;</td>
</tr>
<tr>
<td>Write(Y);</td>
<td>Write(Y);</td>
<td>Write(Y);</td>
</tr>
<tr>
<td>commit;</td>
<td>commit;</td>
<td>commit;</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>Read(X);</td>
<td>Read(X);</td>
<td>Read(X);</td>
</tr>
<tr>
<td>$X:=X-1$;</td>
<td>$X:=X-1$;</td>
<td>$X:=X-1$;</td>
</tr>
<tr>
<td>Write(X);</td>
<td>Write(X);</td>
<td>Write(X);</td>
</tr>
<tr>
<td>Read(Y);</td>
<td>Read(Y);</td>
<td>Read(Y);</td>
</tr>
<tr>
<td>$Y=Y+20$;</td>
<td>$Y=Y+20$;</td>
<td>$Y=Y+20$;</td>
</tr>
<tr>
<td>Write(Y);</td>
<td>Write(Y);</td>
<td>Write(Y);</td>
</tr>
<tr>
<td>commit;</td>
<td>commit;</td>
<td>commit;</td>
</tr>
</tbody>
</table>
Consider $S_1$ and $S_2$ for Transactions $T_1$ and $T_2$.

If $X = 10$ and $Y = 20$

- After $S_1$ $X = 7$ and $Y = 28$
- After $S_2$ $X = 7$ and $Y = 27$

**Schedule $S_1$**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(X); X:=X− 2; Write(X); Read(Y); Y = X + 20; Write(Y); commit;</td>
<td>Read(X); X:=X− 2; Write(X); commit;</td>
</tr>
</tbody>
</table>

**Schedule $S_2$**

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(X); X:=X− 1; Write(X); commit;</td>
<td>Read(X); X:=X− 1; Write(X); commit;</td>
</tr>
</tbody>
</table>

A Schedule is Serializable if it Matches Either $S_1$ or $S_2$, Even if $S_1$ and $S_2$ Produce Different Results!
Serializability Theorem

● A Dependency Exists Between Two Transactions If:
  – They Access the Same Data Item Consecutively in the Schedule and One of the Accesses is a Write

● Three Cases: $T_2$ Depends on $T_1$, Denoted by $T_1 \rightarrow T_2$
  – $T_2$ Executes a Read(x) after a Write(x) by $T_1$
  – $T_2$ Executes a Write(x) after a Read(x) by $T_1$
  – $T_2$ Executes a Write(x) after a Write(x) by $T_1$
A Precedence Graph of a Schedule is a Graph \( G = <TN, DE> \), where

- Each Node is a Single Transaction;
  i.e., \( TN = \{T_1, ..., T_n\} \) (\( n > 1 \))

- And

  - Each Arc (Edge) Represents a Dependency Going from the
    Preceding Transaction to the Other
  i.e., \( DE = \{e_{ij} | e_{ij} = (T_i, T_j), T_i, T_j \in TN\} \)

  - Use Dependency Cases on Prior Slide

The Serializability Theorem

- A Schedule is *Serializable* if and only of its Precedence Graph is Acyclic
Consider $S_1$ and $S_2$ for Transactions $T_1$ and $T_2$

Consider the Two Precedence Graphs for $S_1$ and $S_2$

No Cycles in Either Graph!

Schedule $S_1$:
- $T_1$: Read($X$); $X := X - 2$; Write($X$);
- $T_2$: Read($Y$); $Y := Y + 20$; Write($Y$); commit;

Schedule $S_2$:
- $T_1$: Read($X$); $X := X - 1$; Write($X$); commit;
- $T_2$: Read($X$); $X := X - 2$; Write($X$);
- Read($Y$); $Y := Y + 20$; Write($Y$); commit;
For $S_3$

- $T_1 \rightarrow T_2$ ($T_2$ Write($X$) After $T_1$ Write($X$))
- $T_2 \rightarrow T_1$ ($T_1$ Write($X$) After $T_2$ Read ($X$))

For $S_4$ $T_1 \rightarrow T_2$ ($T_2$ Read/Write($X$) After $T_1$ Write($X$))

Schedule $S_3$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read($X$); $X := X - 2$;</td>
<td>Read($X$); $X := X - 1$;</td>
</tr>
<tr>
<td>Write($X$); Read($Y$);</td>
<td>Write($X$);</td>
</tr>
<tr>
<td>$Y = Y + 20$; commit;</td>
<td></td>
</tr>
</tbody>
</table>

Schedule $S_4$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read($X$); $X := X - 2$;</td>
<td>Read($X$); $X := X - 1$;</td>
</tr>
<tr>
<td>Write($X$);</td>
<td>Write($X$);</td>
</tr>
<tr>
<td>Read($Y$); commit;</td>
<td>Write($Y$); commit;</td>
</tr>
<tr>
<td>$Y = Y + 20$; Write($Y$); commit;</td>
<td>$Y = Y + 20$; Write($Y$); commit;</td>
</tr>
</tbody>
</table>
Transaction Support in SQL

- A single SQL statement is always considered to be atomic. Either the statement completes execution without error or it fails and leaves the database unchanged.

- With SQL, there is no explicit Begin Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered.

- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.
Characteristics specified by a SET TRANSACTION statement in SQL:

- **Access mode:** READ ONLY or READ WRITE. The default is READ WRITE unless the isolation level of READ UNCOMMITTED is specified, in which case READ ONLY is assumed.

- **Diagnostic size** $n$, specifies an integer value $n$, indicating the number of conditions that can be held simultaneously in the diagnostic area. (Supply user feedback information)
Isolation level <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE. With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability. However, if any transaction executes at a lower level, then serializability may be violated.
Potential problem with lower isolation levels:

- **Dirty Read**: Reading a value that was written by a transaction which failed.

- **Nonrepeatable Read**: Allowing another transaction to write a new value between multiple reads of one transaction.

A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value. Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.
**Phantoms**: New rows being read using the same read with a condition.

A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause. Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1. If T1 is repeated, then T1 will see a row that previously did not exist, called a **phantom**.
Possible violation of serializability:

Type of Violation

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty read</th>
<th>nonrepeatable read</th>
<th>phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Sample SQL transaction:

EXEC SQL whenever sqlerror go to UNDO;
EXEC SQL SET TRANSACTION
  READ WRITE
  DIAGNOSTICS SIZE 5
  ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
  INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
  VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
  SET SALARY = SALARY * 1.1
  WHERE DNO = 2;
EXEC SQL COMMIT;
  GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ...