Transaction Management and Concurrency

CS 341 Database Systems

ACCESS TO THE DATABASE

- Large databases are usually shared by many users, and resources
- It is efficient to allow concurrent access.

Transaction

- "An action, or series of actions, carried out by a single user or application program, which **reads** or **updates** the contents of the **database**."

Transaction

- A transaction is a **logical unit of work**, as well as **unit of recovery**.
- It is broken down into a sequence of atomic operations, which if any fail, the whole transaction is undone.

```
SELECT | INSERT | …
          …
          … work …
          …
COMMIT | ROLLBACK
```

Transactions

- Series of database commands with clear semantics
  - e.g. transfer of funds from one account to another
- **Commit**:
  - If nothing fails, **commit** point where the DB should be consistent.
  - All updates are **tentative** until committed.
- **Rollback:**
  - If any command fails → whole series of commands is undone.
- Any DBMS support these (and language e.g. SQL)

Transaction

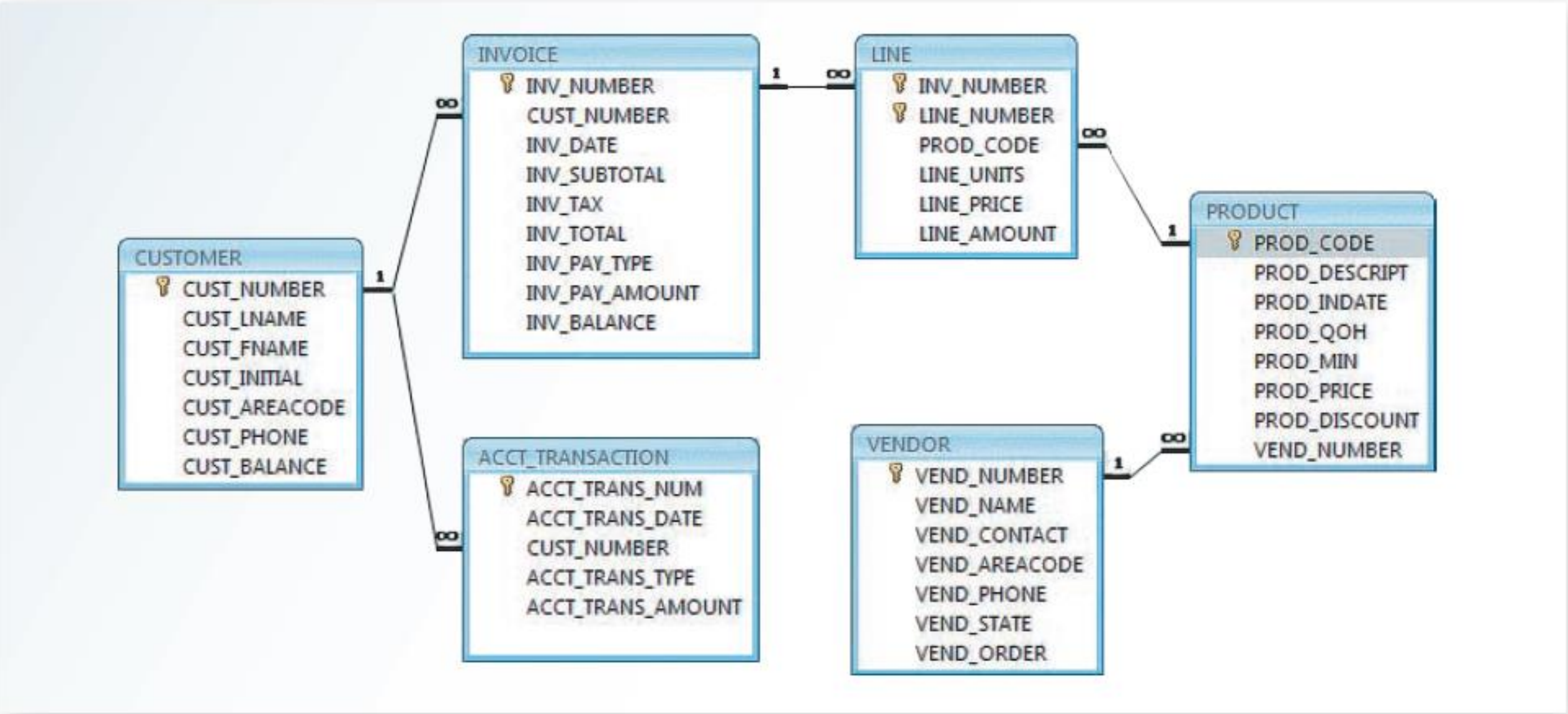
A transaction is any action that **reads** from and/or **writes** to a database.

A transaction may consist of the following:

- A simple **SELECT** statement to generate a list of table contents.
- A series of related **UPDATE** statements to change the values of attributes in various tables.
- A series of **INSERT** statements to add rows to one or more tables.
- A combination of **SELECT**, **UPDATE**, and **INSERT** statements.

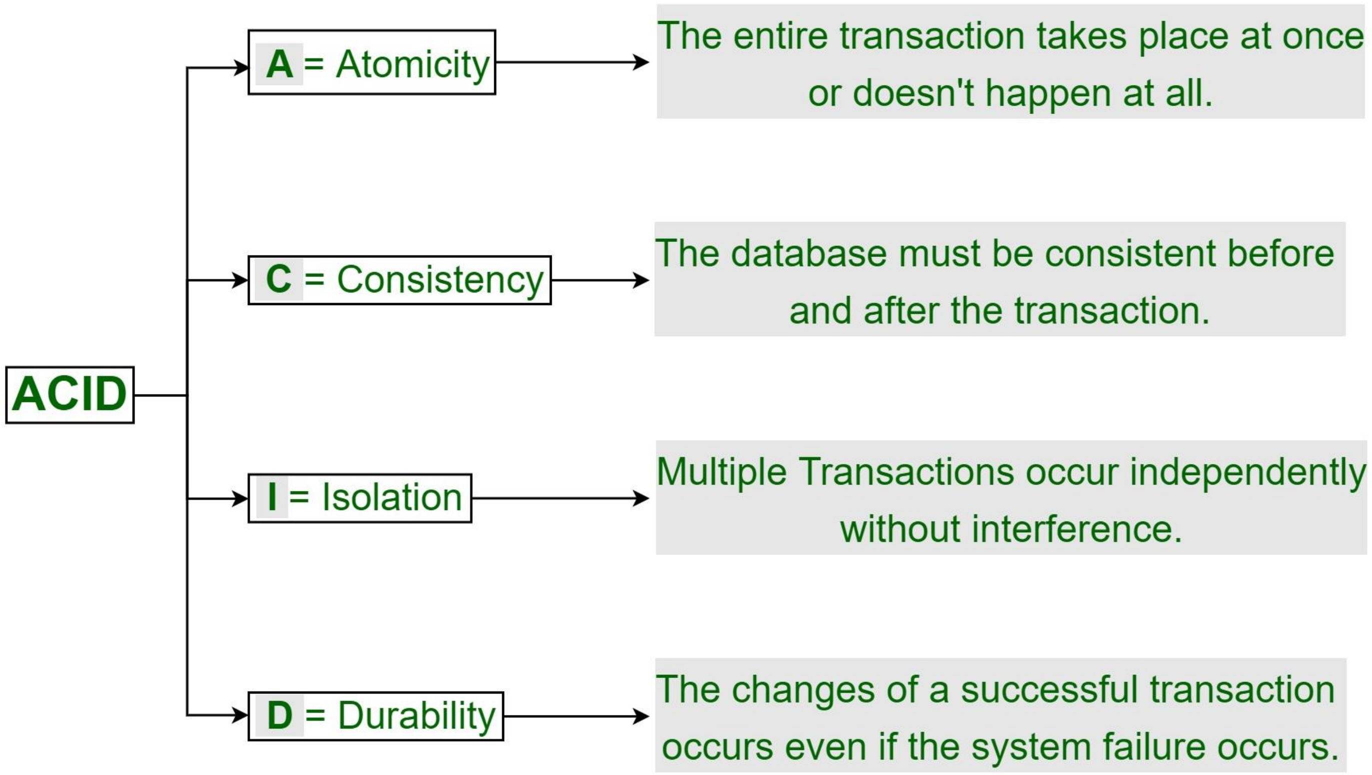
- Suppose that you sell a product to a customer and the customer may charge the purchase to his or her account.
- The sales transaction consists of at least the following parts:
  - Write a new customer invoice.
  - Reduce the quantity on hand in the product's inventory.
  - Update the account transactions.
  - Update the customer balance.

Figure 10.1 The Ch10_SaleCo Database Relational Diagram



```sql
INSERT INTO INVOICE
        VALUES (1009, 10016, '18-Jan-2022', 256.99, 20.56, 277.55, 'cred', 0.00, 277.55);
INSERT INTO LINE
        VALUES (1009, 1, '89-WRE-Q', 1, 256.99, 256.99);

UPDATE  PRODUCT
SET     PROD_QOH = PROD_QOH - 1
WHERE   PROD_CODE = '89-WRE-Q';

UPDATE  CUSTOMER
SET     CUST_BALANCE = CUST_BALANCE + 277.55
WHERE   CUST_NUMBER = 10016;

INSERT INTO ACCT_TRANSACTION
        VALUES (10007, '18-Jan-22', 10016, 'charge', 277.55);

COMMIT;
```

ACID Properties

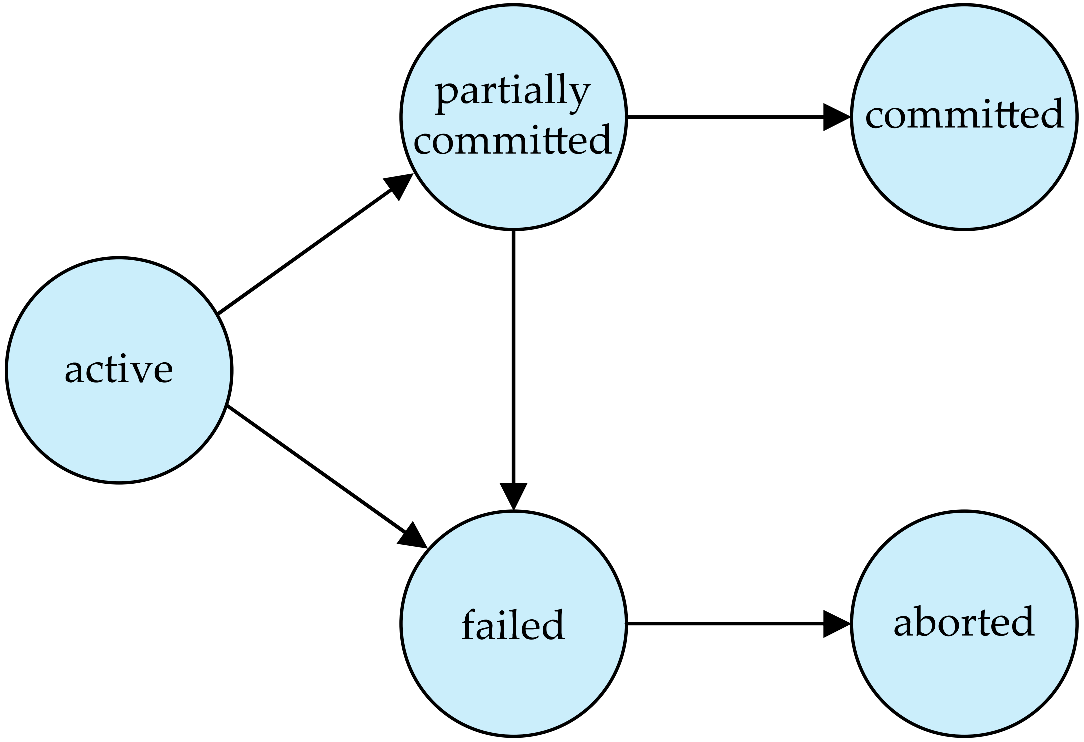


ACID Properties

- **Atomicity**: all or nothing (any error → Rollback, as if nothing happened)
- **Consistency:** a consistent state always leads to another consistent state
- **Isolation:** a transaction's updates are hidden until it Commits
- **Durability:** after a Commit, updates persist

- These are the **ACID** properties of transactions.

Transaction State



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.

System Recovery

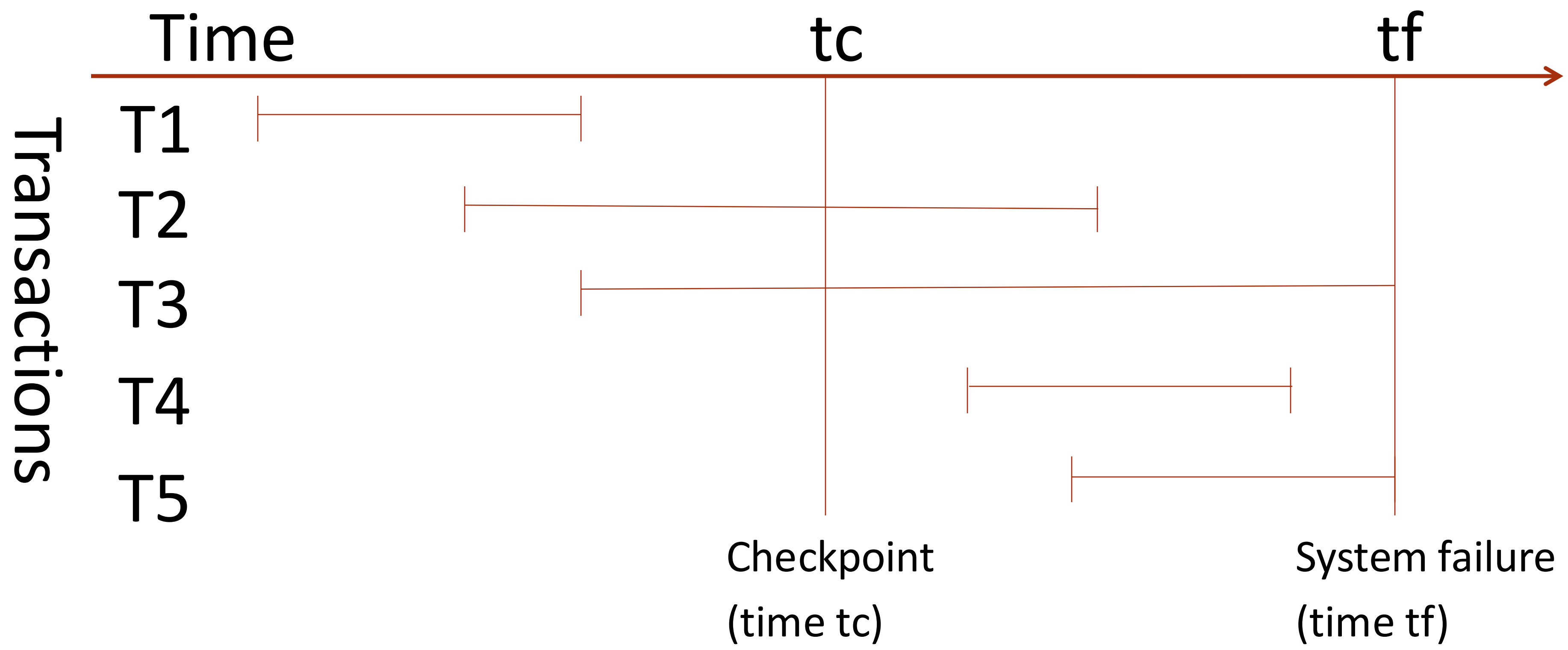
- How does the system recover after a system failure (e.g., power failure) or media failure (e.g., disk crash)?
- In the event of a crash …
    - Contents of main memory are lost.
    - Transaction log persists.
    - At failure, certain transactions will be complete while others partially complete.
- Note that updates are held in memory buffers and written out periodically.

Recovery

To recover the state of the database we can use:

- A log file recording every database operation.
- Checkpoints recording the state of all active transactions.
  - Then: develop an algorithm for transactions to **UNDO**,
  - and those that we need to **REDO**, to effect recovery.
  - at intervals, the system will:
    - Flush its buffers – buffers are forced to write changes to secondary storage.
    - Write out a checkpoint record to log indicating which transactions are in *progress*.

Five Transaction categories



- The most recent check point record was taken at time tc.

Transactions

Completed		Un-Finished
Cached	Written	
T2 T4	T1	T3 T5

Transactions

- CHECKPOINT RECORD:
  - T3, T5 : **undone** (rollback possible)
  - T2, T4 : **re-done**

- DBMS creates REDO/UNDO list from checkpoint record + system log.

- **Isolation**
  - order of recovery *not* crucial,
  - only DB should be consistent near tf (time of failure)

Concurrency

Concurrency

- Many users hence many transactions - at the same time.
- Databases are **shared**!

- So: Transactions must be ***isolated*** => need of **concurrency control** to ensure no interference.
- We will look at:
  - **3 classic problems** on concurrent access
  - **Locking** mechanism
  - **Deadlock** resolution

Three classic problems

Problem: *Two or more transactions read / write on the same part of the database.*

Although transactions execute correctly, results may **interleave** in different ways leading to the 3 classic problems.

- *Lost Update*
- *Uncommitted Dependency*
- *Inconsistent Analysis*

Lost Update problem

Time	User 1 (Trans A)	User2 (Trans B)
1	Retrieve t	
2		Retrieve t
3	Update t	
4		Update t
5		
6		
7		

t : tuple in a table.
Trans A loses an update at t4.
The update at t3 is lost (overwritten) at t4 by B.

Lost Update problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	200 $bal_x = bal_x + 100$	100
t_4	90 $bal_x = bal_x - 10$	200 write(bal_x)	200
t_5	90 write(bal_x)	commit	90
t_6	commit		90

- An apparently successfully completed update operation by one user can be overridden by another user. This is known as the lost update problem

Serial VS Concurrent

Table 10.3 Serial Execution of Two Transactions

Time	Transaction	Step	Stored Value
1	T1	Read PROD_QOH	35
2	T1	PROD_QOH = 35 + 100	
3	T1	Write PROD_QOH	135
4	T2	Read PROD_QOH	135
5	T2	PROD_QOH = 135 − 30	
6	T2	Write PROD_QOH	105

Table 10.4 Lost Updates

Time	Transaction	Step	Stored Value
1	T1	Read PROD_QOH	35
2	T2	Read PROD_QOH	35
3	T1	PROD_QOH = 35 + 100	
4	T2	PROD_QOH = 35 − 30	
5	T1	Write PROD_QOH (lost update)	135
6	T2	Write PROD_QOH	5

Uncommitted Dependency

- 2 Problems (T1-3 ; T6-8).
- One trans is allowed to retrieve/update) a tuple updated by another, but not yet committed.
- Trans A is dependent at time t2 on an **uncommitted change** made by Trans B, which is lost on Rollback.

Time	User 1 (Trans A)	User 2 (Trans B)
1		Update t
2	Retrieve t	
3		Rollback
4		
5		
6		Update t
7	Update t	
8		Rollback

Uncommitted Dependency

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

- The uncommitted dependency problem occurs when one transaction is allowed to see the intermediate results of another transaction before it has committed.

Table 10.5 Transactions Creating an Uncommitted Data Problem

Transaction	Computation
T1: Purchase 100 units	PROD_QOH = PROD_QOH + 100 (Rolled back)
T2: Sell 30 units	PROD_QOH = PROD_QOH – 30

Table 10.6 Correct Execution of Two Transactions

Time	Transaction	Step	Stored Value
1	T1	Read PROD_QOH	35
2	T1	PROD_QOH = 35 + 100	
3	T1	Write PROD_QOH	135
4	T1	*****ROLLBACK *****	35
5	T2	Read PROD_QOH	35
6	T2	PROD_QOH = 35 – 30	
7	T2	Write PROD_QOH	5

Table 10.7 An Uncommitted Data Problem

Time	Transaction	Step	Stored Value
1	T1	Read PROD_QOH	35
2	T1	PROD_QOH = 35 + 100	
3	T1	Write PROD_QOH	135
4	T2	Read PROD_QOH (Read uncommitted data)	135
5	T2	PROD_QOH = 135 – 30	
6	T1	***** ROLLBACK *****	35
7	T2	Write PROD_QOH	105

Inconsistent Analysis

Initially: Acc 1 = 40; Acc2 = 50; Acc3 = 30;

Time	User 1 (Trans A)	User 2 (Trans B)
1	Retrieve Acc 1 : Sum = 40	
2	Retrieve Acc2 : Sum = 90	
3		Retrieve Acc3 :
4		Update Acc3: 30 → 20
5		Retrieve Acc1:
6		Update Acc1: 40 → 50
7		commit
8	Retrieve Acc3: Sum = 110 (not 120)	

- Trans A sees inconsistent DB state after B updated Accumulator => performs inconsistent analysis.

Inconsistent Analysis

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

The problem of inconsistent analysis occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.

Inconsistent Analysis Eg.2

Table 10.8 Retrieval during Update

Transaction T1	Transaction T2
SELECT SUM(PROD_QOH) FROM PRODUCT	UPDATE PRODUCT SET PROD_QOH = PROD_QOH + 10 WHERE PROD_CODE = 1546-QQ2
	UPDATE PRODUCT SET PROD_QOH = PROD_QOH − 10 WHERE PROD_CODE = 1558-QW1
	COMMIT;

Table 10.9 Transaction Results: Data Entry Correction

	Before	After
PROD_CODE	PROD_QOH	PROD_QOH
11QER/31	8	8
13-Q2/P2	32	32
1546-QQ2	15	(15 + 10) → 25
1558-QW1	23	(23 − 10) → 13
2232-QTY	8	8
2232-QWE	6	6
Total	**92**	**92**

Inconsistent Analysis Eg.2

Table 10.10 Inconsistent Retrievals

Time	Transaction	Action	Value	Total
1	T1	Read PROD_QOH for PROD_CODE = '11QER/31'	8	8
2	T1	Read PROD_QOH for PROD_CODE = '13-Q2/P2'	32	40
3	T2	Read PROD_QOH for PROD_CODE = '1546-QQ2'	15	
4	T2	PROD_QOH = 15 + 10		
5	T2	Write PROD_QOH for PROD_CODE = '1546-QQ2'	25	
6	T1	Read PROD_QOH for PROD_CODE = '1546-QQ2'	25	(After) 65
7	T1	Read PROD_QOH for PROD_CODE = '1558-QW1'	23	(Before) 88
8	T2	Read PROD_QOH for PROD_CODE = '1558-QW1'	23	
9	T2	PROD_QOH = 23 − 10		
10	T2	Write PROD_QOH for PROD_CODE = '1558-QW1'	13	
11	T2	***** COMMIT *****		
12	T1	Read PROD_QOH for PROD_CODE = '2232-QTY'	8	96
13	T1	Read PROD_QOH for PROD_CODE = '2232-QWE'	6	102

Read Write Conflicts

Table 10.11 Read/Write Conflict Scenarios: Conflicting Database Operations Matrix

	Transactions		
	T1	**T2**	**Result**
Operations	Read	Read	No conflict
	Read	Write	Conflict
	Write	Read	Conflict
	Write	Write	Conflict

Why these problems occur?

- Retrieve : ‘read’ (R)
- Update : ‘write’ (W).
- Interleaving two transactions:

RR	No problem
WW	Lost update
WR	Uncommitted dependency
RW	Inconsistent analysis

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	$\boxed{\text{write}(bal_x)}$	200
t_5	$\boxed{\text{write}(bal_x)}$	commit	90
t_6	commit		90

Lost Update **WW**

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	$\boxed{\text{write}(bal_x)}$	200
t_5	$\boxed{\text{read}(bal_x)}$	:	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
	commit		190

Uncommitted Dependency **WR**

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	$\boxed{\text{read}(bal_x)}$	100	50	25	0
t_4	$bal_x = bal_x - 10$	$sum = sum + bal_x$	100	50	25	100
t_5	$\boxed{\text{write}(bal_x)}$	read(bal_y)	90	50	25	100
t_6	read(bal_z)	$sum = sum + bal_y$	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		$sum = sum + bal_z$	90	50	35	185
t_{11}		commit	90	50	35	185

Inconsistent Analysis **RW**

Session 02

ANSI/ISO Transaction Isolation Levels

- Transaction isolation levels refer to the degree to which transaction data is "**protected or isolated**" from other ***concurrent*** transactions.
- The isolation levels are described based on what data other transactions can see (read) during execution. More precisely, the *transaction isolation levels are described by the type of "reads" that a transaction allows or does not allow.*

Types of Read Operations

1. **Dirty reads**
   - A transaction reads data that has been written by another transaction that has not been committed yet.
2. **Nonrepeatable (fuzzy) reads**
   - A transaction re-reads data it has previously read and finds that another committed transaction has *modified or deleted* the data. For example, a user queries a row and then later queries the same row, only to discover that the data has changed.

Types of Read Operations

3. Phantom Reads

- A transaction re-runs a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.
- For example, a transaction queries the number of employees. Five minutes later it performs the same query, but now the number has increased by one because another user inserted a record for a new hire. More data satisfies the query criteria than before, but unlike in a fuzzy read the previously read data is unchanged.

Isolation Levels

- **Read Uncommitted** will read uncommitted data from other transactions.
  - The database **does not place any locks on the data**, which increases transaction performance but at the cost of data consistency.

- **Read Committed** forces transactions to read only committed data.
  - **Default mode** of operation for most databases (incl. Oracle and SQL Server).
  - The database will **use exclusive locks on data**, causing other transactions to **wait** until the original transaction commits.

Isolation Levels

- The **Repeatable Read** isolation level ensures that queries return consistent results.
  - Uses **shared locks** - ensure other transactions do not update a row after the original query reads it.
  - However, new rows are read **(phantom read)** as these rows did not exist when the first query ran.

- The **Serializable** isolation level is the most restrictive level defined by the ANSI SQL standard.
  - Deadlocks are always possible.
  - Most databases use a deadlock detection approach to transaction management, and, therefore, they will detect "deadlocks" during the transaction validation phase and reschedule the transaction.

Preventable Read Phenomena by Isolation Level

Table 10.15 Transaction Isolation Levels

	Isolation Level	Allowed			Comment
		Dirty Read	Nonrepeatable Read	Phantom Read	
Less restrictive	Read Uncommitted	Y	Y	Y	Reads uncommitted data, and allows nonrepeatable reads and phantom reads.
	Read Committed	N	Y	Y	Does not allow uncommitted data reads but allows nonrepeatable reads and phantom reads.
	Repeatable Read	N	N	Y	Only allows phantom reads.
More restrictive	Serializable	N	N	N	Does not allow dirty reads, nonrepeatable reads, or phantom reads.
Oracle/SQL Server Only	Read Only/Snapshot	N	N	N	Supported by Oracle and SQL Server. The transaction can only see the changes that were committed at the time the transaction started.

Purpose of Isolation Levels

- The reason for the different levels of isolation is to increase transaction concurrency.
- The isolation levels go from the least restrictive (*Read Uncommitted*) to the more restrictive (*Serializable*).
- The **higher the isolation level the more locks (*shared and exclusive*) are required** to improve data consistency, at the expense of transaction concurrency performance.

Setting Isolation Levels in DBMS

- Set at transaction level:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Set at session level:

```
ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;
ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED;
```

- MySQL and SQL Server support all four ANSI isolation levels.
- Oracle supports READ COMMITED and SERIALIZABLE (Oracle would suggest that it has superior implementation of the other two isolation levels. By default, set to read committed)

Locking Protocol

- **Locking Protocol**
  - Other approaches : *serializability, time-stamping, and shadow-paging.*
- If risk of interference is low => **Two-Phase Locking** ~ common approach *although it requires deadlock avoidance!!*
- A transaction can lock a database item from being updated (by another transaction) while it is being used. It acquires a lock on the item of interest and releases it when it has finished.
- Lock applies to a tuple :
  - exclusive (write; X)
  - shared(read; S)

Exclusive VS Shared Locks

Exclusive Lock

- An **exclusive lock** exists when access is reserved specifically for the transaction that locked the object.
- The exclusive lock must be used when the potential for conflict exists.
- An exclusive lock is issued when a transaction wants ***to update (write)*** a data item and *no locks are currently held on that data item by any other transaction.*

Shared Lock

- A **shared lock** exists when concurrent transactions are granted read access on the basis of a common lock.
- No conflict as long as all the concurrent transactions are read-only.
- A shared lock is issued when a transaction wants to **read data** from the database and *no exclusive lock is held on that data item.*

Problem: Lost Update

Time	User 1 (Trans A)	User2 (Trans B)
1	Retrieve t	
2		Retrieve t
3	Update t	
4		Update t
5		
6		
7		

t : tuple in a table.
Trans A loses an update at t4.
The update at t3 is lost (overwritten) at t4 by B.

Solved: Lost Update

Time	User 1 (Trans A)	User2 (Trans B)
1	Retrieve t (get S-lock on t)	
2		Retrieve t (get S-lock on t)
3	Update t (request X-lock on t)	
4	wait	Update t (request X-lock on t)
5	wait	wait
6	wait	wait
7		

- No update lost but → Deadlock

Problem:
Uncommitted Dependency

- 2 Problems (T1-3 ; T6-8).
- One trans is allowed to retrieve/update) a tuple updated by another, but not yet committed.
- Trans A is dependent at time t2 on an **uncommitted change** made by Trans B, which is lost on Rollback.

Time	User 1 (Trans A)	User 2 (Trans B)
1		Update t
2	Retrieve t	
3		Rollback
4		
5		
6		Update t
7	Update t	
8		Rollback

Solved: Uncommitted Dependency

Time	User 1 (Trans A)	User 2 (Trans B)
1		Update t (get X-lock on t)
2	Retrieve t (request S-lock on t)	-
3	wait	-
4	wait	-
5	wait	Commit / Rollback (releases X-lock on t)
6	Resume: Retrieve t (get S-lock on t)	
7	-	
8		

Problem: Inconsistent Analysis

- Trans A sees inconsistent DB state after B updated Accumulator => performs inconsistent analysis.

Initially: Acc 1 = 40;   Acc2 = 50;   Acc3 = 30;

Time	User 1 (Trans A)	User 2 (Trans B)
1	Retrieve Acc 1 : Sum = 40	
2	Retrieve Acc2 : Sum = 90	
3		Retrieve Acc3 :
4		Update Acc3: 30 → 20
5		Retrieve Acc1:
6		Update Acc1: 40 → 50
7		commit
8	Retrieve Acc3: Sum = 110 (not 120)	

Solved: Inconsistent Analysis

Time	User 1 (Trans A)	User 2 (Trans B)
1	Retrieve Acc1 : (get S-lock) Sum = 40	
2	Retrieve Acc2 : (get S-lock) Sum = 90	
3		Retrieve Acc3: (get S-lock)
4		Update Acc3: (get X-lock) 30 → 20
5		Retrieve Acc1: (get S-lock)
6		Update Acc1: (request X-lock) wait
7	Retrieve Acc3: (request S-lock) wait	wait wait wait

- Inconsistent Analysis is prevented → Deadlock