Chapter 9 – Transaction Processing Concepts

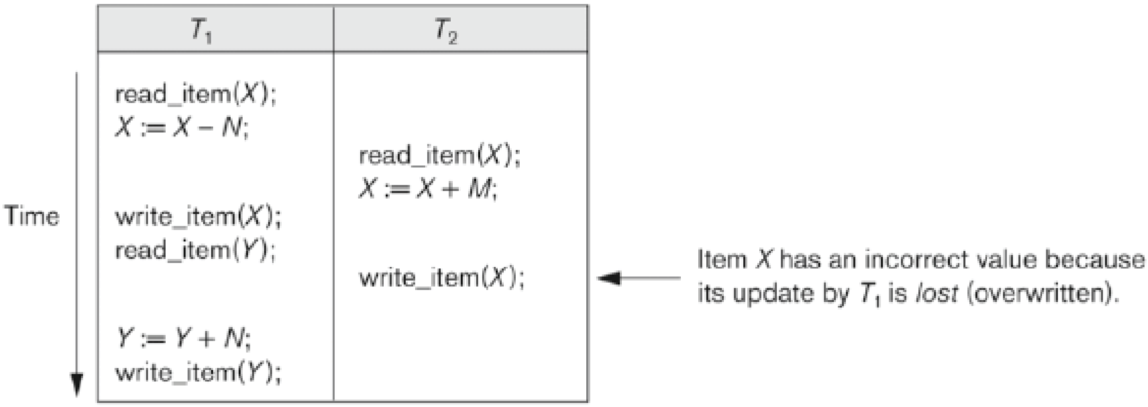
- Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions.
- Transaction is a logical unit of database processing that includes one or more database access operations.
- Basic database access operations that a transaction can include are:
  - **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. It 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)**: Writes the value of program variable X into the database item named X. It 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).

Why Concurrency control is needed

When concurrent transactions execute in an uncontrolled manner, several problems can occur.

1. 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. Suppose that transactions T1 and T2 are submitted at same time, and their operations are interleaved as shown below.



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

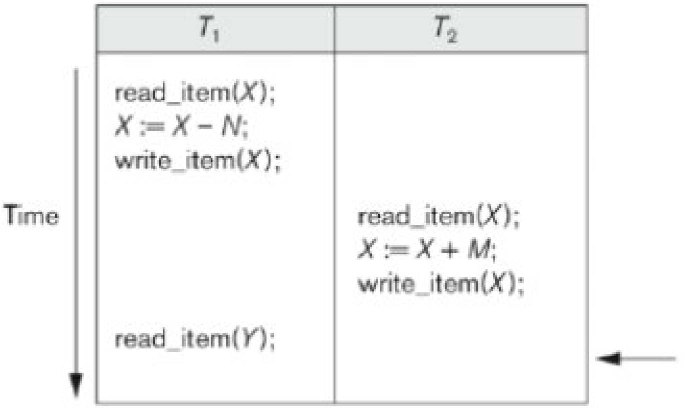
Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

The final value of X is incorrect, because T2 reads the value of X before T1 changes it in the database, and hence updated value resulting from T1 is lost. For example, if X= 80 at the start (originally there were 80 reservations on the flight), N = 5(T1 transfers 5 seat reservations from flight corresponding to X to the flight corresponding to Y), and M = 4(T2 reserves 4 seats on X), the final result should be X = 79; But it is X = 84 because the update in T1 was lost.

2. **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 following figure shows an example where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, T2 reads the "temporary" value of X, which will not be recorded permanently in the database because of the failure of T1. The value of item X that is read by T2 is called **dirty data.**



T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

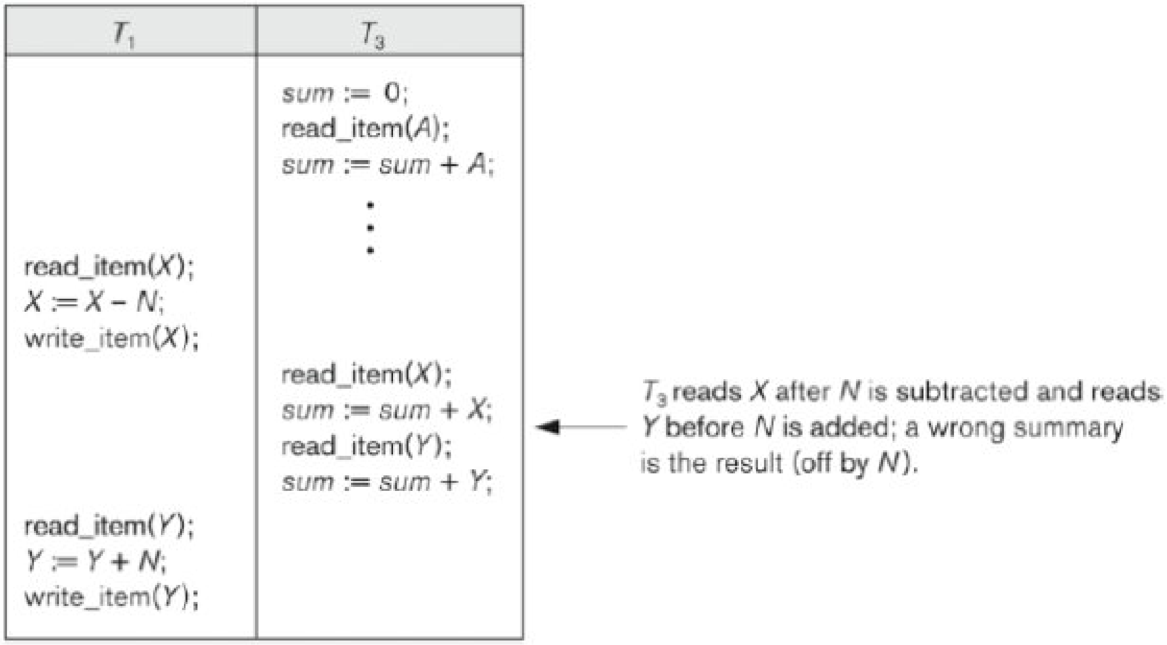
read_item(Y);	

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.

3. **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.

For example, suppose that a transaction T3 is calculating the total number of reservations on all flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in the above figure occurs, the result of T3 will be off by an amount N because T3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.



T_1	T_3

	sum := 0;
	read_item(A);
	sum := sum + A;
	⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	sum := sum + X;
	read_item(Y);
	sum := sum + Y;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

4. **Unrepeatable Read problem**

   Transaction A reads the value of a data item multiple times during the transaction's life, however, transaction B is allowed to update the data item in between A's reads. Hence, A receives different values for its reads of the same item. (Some sources also call this the repeatable read problem. The definition is the same.)

What causes a Transaction to fail?

1. A computer failure (system crash):
A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error:
Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

3. Local errors or exception conditions detected by the transaction:
Certain conditions necessitate cancellation of the transaction. For example, 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.

4. 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

5. 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.

6. 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.

Transaction States

At any instance of time a transaction will be in any of the following states

- Active: This is the initial state, the transaction stays in this state while it is executing and it can issue READ, WRITE operation in this state
- Partially committed: After the final statement has been executed, transaction moves to this state.
- Committed: After successful completion, the transaction moves to this state.
- Aborted: after the discovery that normal execution can no longer proceed, the transaction will be rolled back and the database has been restored to it state prior to the start of the transaction
- Terminated: This is the final state. The transaction transitions to this state either from committed state or aborted state.

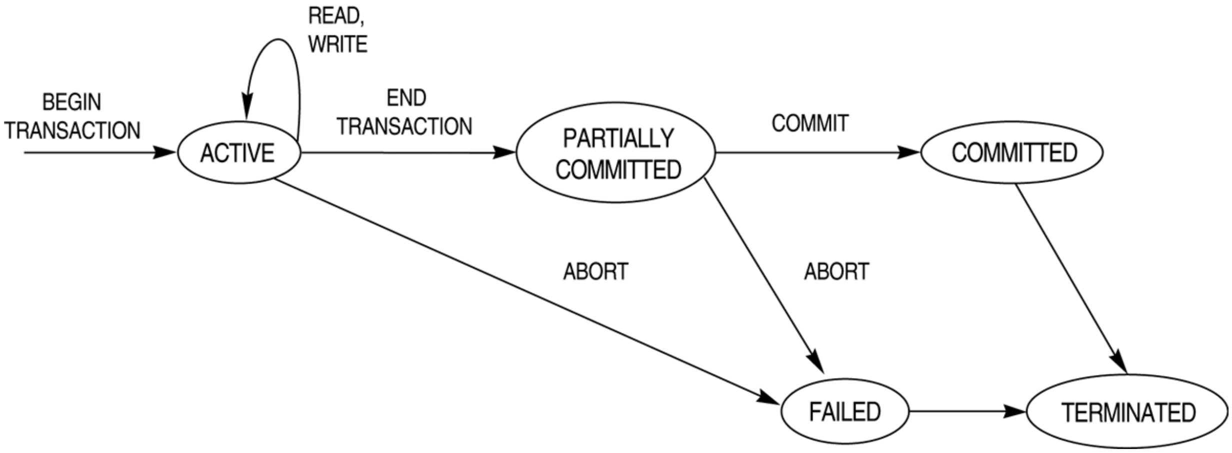


Figure: State transition Diagram

A transaction starts in the active state. When it finishes its final statement, it enters partially committed state. At this point, even the transaction has already completed all its statements, it may have still be aborted since the actual output may still be temporarily residing in main memory not yet be copied to the disk. A transaction is committed only if it has performed updates transforms the database into a consistent sate, which must be persist even if there is a system failure. A transaction is said to have terminated if either committed or aborted.

A transaction enters failed state after the system cannot process the transaction normal execution because of hardware or logical errors. Such transactions must be rolled back and enter aborted state. At this point, the system can do either restart the transaction or kill the transaction.

The system log

In order to recover the database from failures that affect transactions, the database system maintains a log file in which it records all the operations that access the values of data items. A log record can have the following entries

- [start_transaction, T] where T is an unique transaction id
- [write, T, X, old_value, new_value] indicate a write operation in transaction T which changed the value of data item X from old-value to new_value
- [read, T, X] : transaction T read the data item X
- [commit, T]. T is committed
- [abort, T] T is aborted

This log file will be used to do recover of database system. We will discuss more about recovery later.

Commit Point of a Transaction

A transaction is at the commit point if all of its operations are successfully complete and the effects of all operations have been recorded in the log and performed in the database. Beyond the commit point, the transaction writes the record, [commit, T] to the log. If a failure occurs, all transactions with no commit record will have to be rolled back.

ACID properties of a transaction:

1. **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
3. **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
4. **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

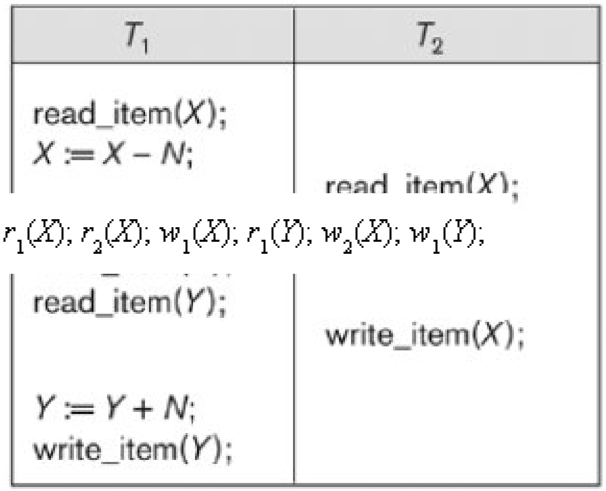
Recall the database state is the set of data and values at a given point in time. A consistent state is one in which the present set of values do not violate constraints on the schema.

Schedules and Recoverability

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

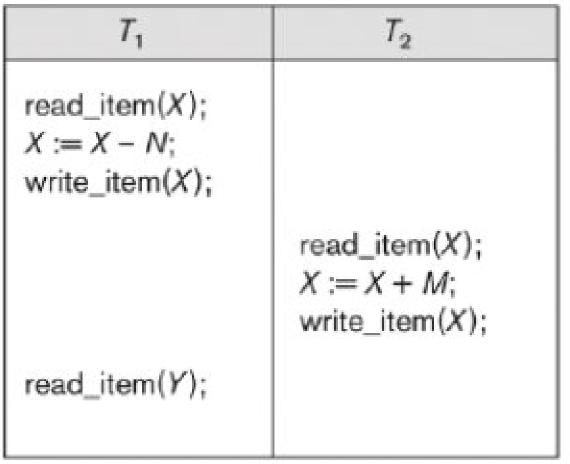
- A schedule S of n transactions T_1, T_2, ..., T_n is an ordering of the operations of the transactions subject to the **constraint** that, for each transaction T_i that participates in S, the operations of T_i in S must appear in the same order in which they occur in T_i.

Sample schedules:



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X);
read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;



T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;

Conflicting Operations

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:

1. They belong to different transactions.
2. They access the same item X
3. At least one of the operations is a write_item(X).

Example:- Consider the following schedule

Sa : r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);

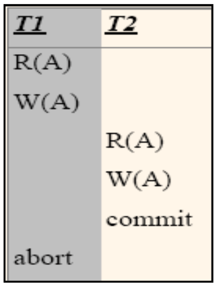
The following operations conflict

- r1(X) and w2(X)
- r2(X) and w1(X)
- w1(X) and w2(X)

Schedules based on Recoverability

Recoverable schedule:

Consider the following schedule,



T1	*T2*
R(A)	
W(A)	
	R(A)
	W(A)
	commit
abort	

In this schedule, Abort of T1 requires abort of T2. But T2 has already committed!

A recoverable schedule is one in which this cannot happen. i.e. a transaction commits only after all the transactions it “depends on” (i.e. it reads from or overwrites) commit.

In a recoverable schedule, no committed transaction ever needs to be rolled back.

Cascadeless Schedule:

With a recoverable schedule it is possible for a phenomenon known as **cascading rollback** (or cascading abort) to occur, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule S_e, where transaction T_2 has to be rolled back because it read item X from T_1, and T_1 then aborted.

S_e: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;

Cascading rollback can be quite time-consuming. A schedule is said to be **cascadeless**, or avoid cascading rollback, if every transaction in the schedule reads only items that were written by *committed* transactions.

Strict Schedule:

A **strict schedule** are composed of transactions that can neither read nor write an item X until the last transaction that wrote X has committed (or aborted). For example, the schedule S_f is a cascadeless schedule but not a strict schedule.

$$S_f: w_1(X, 5); w_2(X, 8); a_1;$$

A *strict* schedule is both *cascadeless* and *recoverable*. A *cascadeless* schedule is a *recoverable* schedule.

Serializability of Schedules

- A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T is executed consecutively in the schedule; otherwise, the schedule is called nonserial.
- The **problem** with serial schedules is that they limit *concurrency* or interleaving of operations.
- A schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n transactions.
- Saying that a *nonserial* schedule S is *serializable* is equivalent to saying that it is correct.
- A **serializable** schedule gives the benefits of *concurrent* execution without giving up any *correctness.*
- Equivalence of two schedules is determined by using any of the following two definitions
  - **Conflict Equivalence**
  - **View Equivalence**

Conflict-Serializable Schedules:

- Two schedules are said to be **conflict equivalent** if the order of any two *conflicting* operations is the same in both schedules.
- A schedule S is **conflict serializable** if it is *conflict equivalent* to some serial schedule S´.

Test for conflict serializability

Let S be a schedule. We construct a directed graph, called a precedence graph from S. This graph consists of a pair G = (V, E), where V is a set of vertices, and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule.

The set of edges consists of all edges Ti → Tj for which one of the following three conditions holds

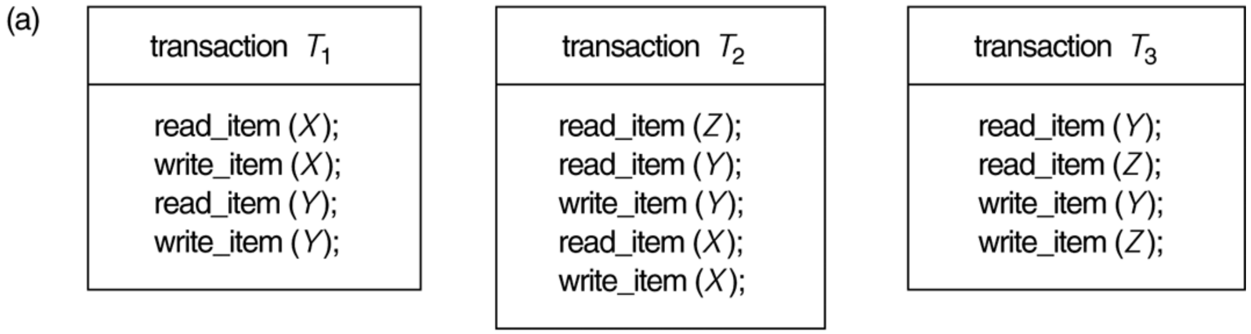
- Ti executes write (Q) before Tj executes read (Q).
- Ti executes read (Q) before Tj executes write (Q).
- Ti executes write (Q) before Tj executes write (Q).

CONFLICT SERIALIZABILITY ALGORITHM:

Testing conflict serializability of a schedule S.

1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
2. For each case in S where Tj executes a read (Q) after Ti executes a write (Q), create an edge (Ti → Tj) in the precedence graph.
3. For each case in S where Tj executes a write (Q) after Ti executes a read (Q), create an edge (Ti→ Tj) in the precedence graph.
4. For each case in S where Tj executes a write (Q) after Ti executes a write (Q), create an edge (Ti→ Tj) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

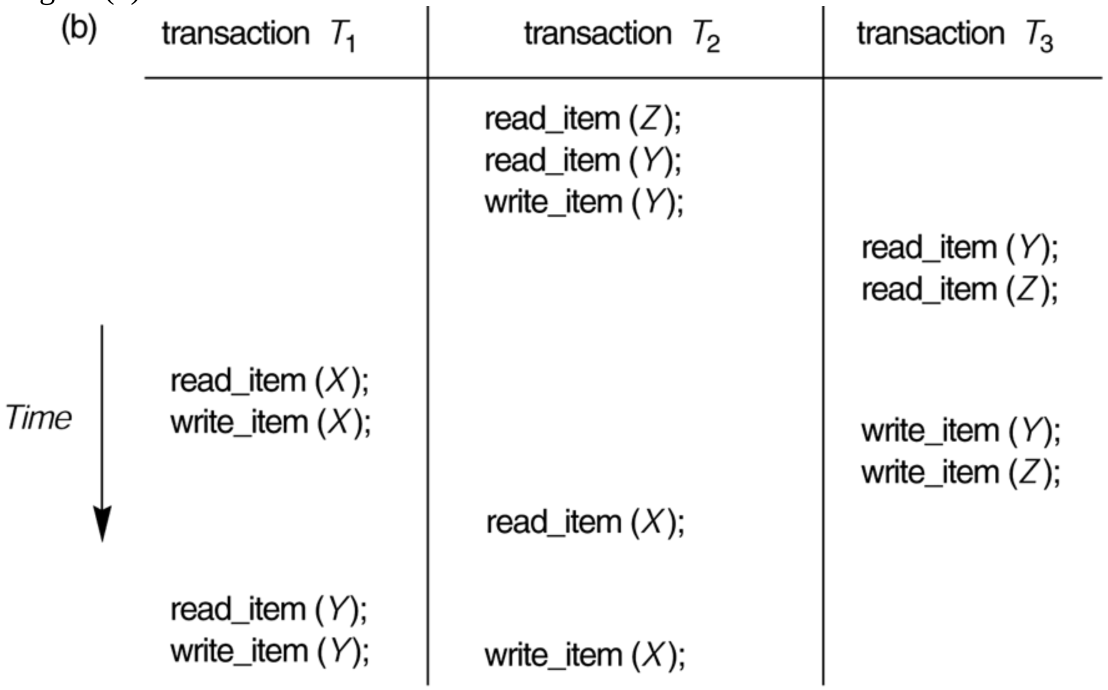
Figure: Example of serializability testing. (a) The READ and WRITE operations of three transactions *T*1, *T*2, and *T*3.


(a)

transaction T_1	transaction T_2	transaction T_3

read_item (X);
write_item (X);
read_item (Y);
write_item (Y);
	read_item (Z);
	read_item (Y);
	write_item (Y);
	read_item (X);
	write_item (X);
		read_item (Y);
		read_item (Z);
		write_item (Y);
		write_item (Z);

Figure (b) Schedule E.



(b)

transaction T_1	transaction T_2	transaction T_3

	read_item (Z);
	read_item (Y);
	write_item (Y);	

		read_item (Y);
		read_item (Z);

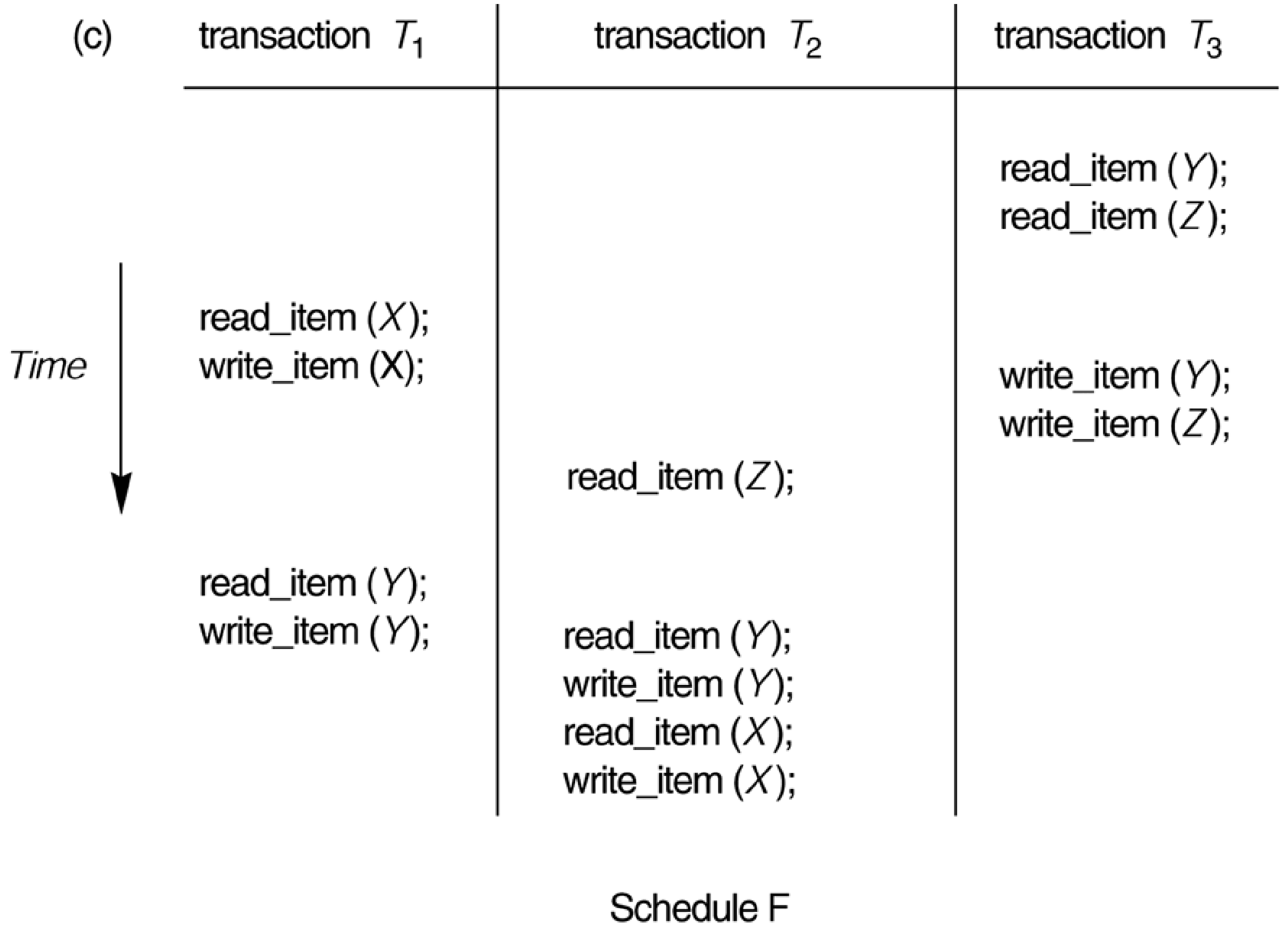
read_item (X);
write_item (X);		

		write_item (Y);
		write_item (Z);

	read_item (X);	

read_item (Y);
write_item (Y);
	write_item (X);	

Schedule E



(c)	transaction T_1	transaction T_2	transaction T_3
			read_item (Y);
			read_item (Z);
	read_item (X);		
	write_item (X);		
			write_item (Y);
			write_item (Z);
		read_item (Z);	
	read_item (Y);		
	write_item (Y);		
		read_item (Y);	
		write_item (Y);	
		read_item (X);	
		write_item (X);	

Schedule F

Figure: (c) Schedule F

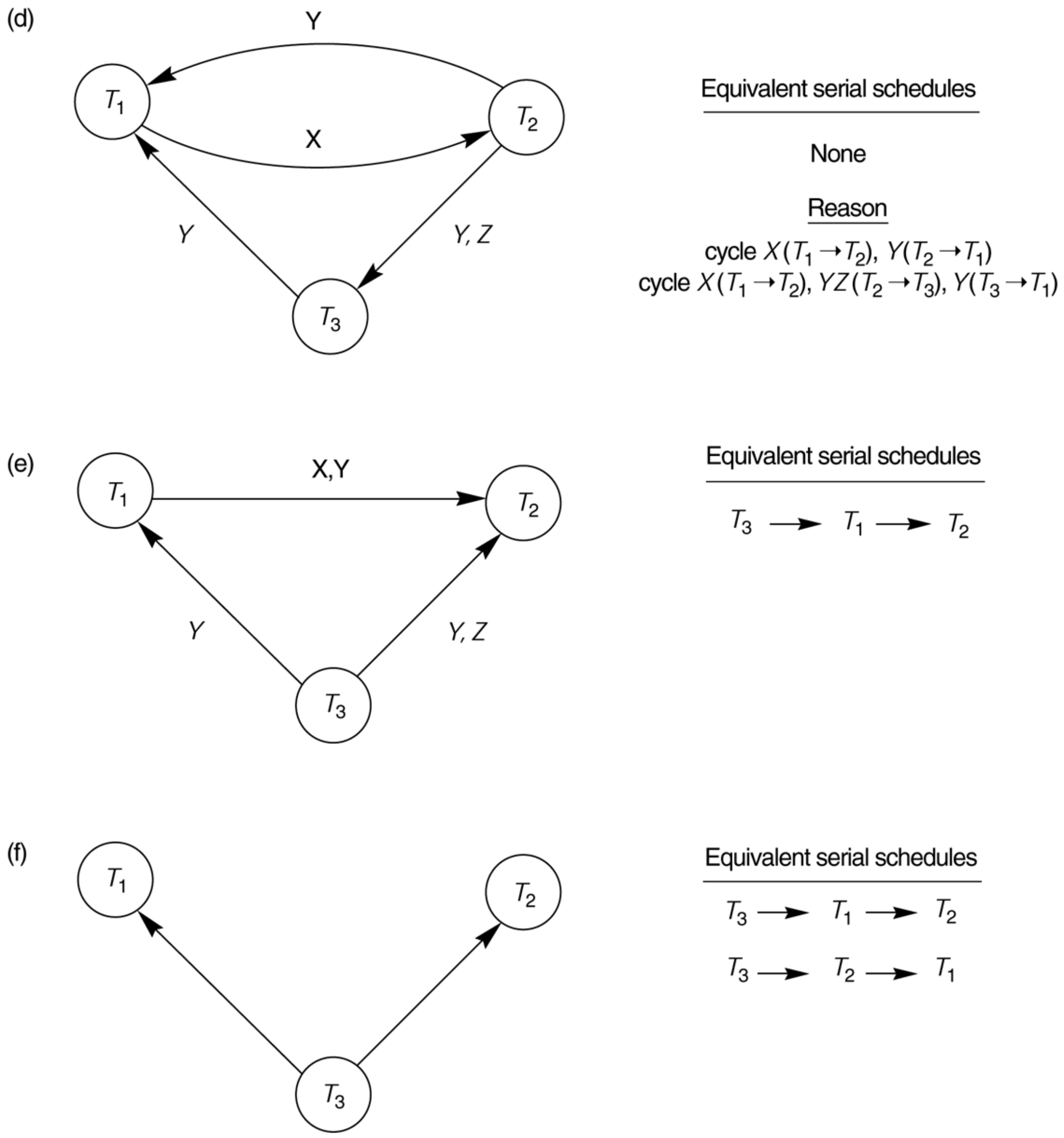

(d)
Y
X
T_1
T_2
Y
Y, Z
T_3
Equivalent serial schedules
None
Reason
cycle $X(T_1 \rightarrow T_2)$, $Y(T_2 \rightarrow T_1)$
cycle $X(T_1 \rightarrow T_2)$, $YZ(T_2 \rightarrow T_3)$, $Y(T_3 \rightarrow T_1)$
(e)
T_1
X,Y
T_2
Y
Y, Z
T_3
Equivalent serial schedules
$T_3 \rightarrow T_1 \rightarrow T_2$
(f)
T_1
T_2
T_3
Equivalent serial schedules
$T_3 \rightarrow T_1 \rightarrow T_2$
$T_3 \rightarrow T_2 \rightarrow T_1$

Figure (d) Precedence graph for schedule *E*.
(e) Precedence graph for schedule *F*.
(f) Precedence graph with two equivalent serial schedules.

View Equivalence and View Serializability

Two schedules S and S' are said to be view equivalent if the following three conditions hold for each data item X:

1. The same set of transactions participate in S and S', and S and S' include the same operations of those transactions.
2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j, the same order between $r_i(X)$ and $w_j(X)$ must hold in S'.
3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S'.

The schedule Sm: r3(Q); w4(Q); w3(Q); w6(Q); c3; c4; c6; is view serializable. It is view equivalent to the serial schedule <T3, T4, T6>, since one read (Q) instruction reads the initial value of Q in both schedules, and T6 performs the final write of Q in both schedules.

Transaction Support in SQL2

- 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.
- Transaction Characteristics can be specified by a SET TRANSACTION statement in SQL2:
  - **Access mode**:
    Access mode of a transaction can be set to READ ONLY or READ WRITE. The default is READ WRITE unless the isolation level of READ UNCOMITTED is specified, in which case READ ONLY is assumed. READ WRITE mode allows update, insert, delete and create commands to be executed. READ ONLY allows data retrieval only.
  - **Diagnostic area size** specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information to the user.
  - **Isolation level :** In database systems, **isolation** is a property that defines how/when the changes made by one operation become visible to other concurrent operations. The isolation levels defined by the ANSI/ ISO SQL standard are:

READ UNCOMMITTED. The READ UNCOMMITTED isolation level allows dirty reads. The basic goal of a READ UNCOMMITTED isolation level is to provide a standards-based definition that allows for nonblocking reads.

READ COMMITTED. The READ COMMITTED isolation level states that a transaction may read only data that has been committed in the database. There are no dirty reads (reads of uncommitted data). There may be nonrepeatable reads (that is, rereads of the same row may return a different answer in the same transaction) and phantom reads (that is, newly inserted and committed rows become visible to a query that weren't visible earlier in the trans-action). READ COMMITTED is perhaps the most commonly used isolation level in database applications everywhere, and it's the default mode for Oracle Database. It's rare to see a different isolation level used in Oracle databases.

REPEATABLE READ. The goal of REPEATABLE READ is to provide an isolation level that gives consistent, correct answers and prevents lost updates. If you have REPEATABLE READ isolation, the results from a given query must be consistent with respect to some point in time. Most databases (not Oracle) achieve repeatable reads through the use of row-level, shared read locks. A shared read lock prevents other sessions from modifying data that you've read. This, of course, decreases concurrency.

SERIALIZABLE. This is generally considered the most restrictive level of transaction isolation, but it provides the highest degree of isolation. A SERIALIZABLE transaction operates in an environment that makes it appear as if there are no other users modifying data in the database. Any row you read is assured to be the same upon a reread, and any query you execute is guaranteed to return the same results for the life of a transaction.

For example, if you execute—

```
select * from T;
begin dbms_lock.sleep( 60*60*24 ); end;
select * from T;
```

—the answers returned from T would be the same, even though you just slept for 24 hours (or you might get an ORA-1555, snapshot too old error). The isolation level assures you these two queries will always return the same results. Side effects, or changes, made by other transactions aren't visible to the query no matter how long it has been running.

Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**
READ UNCOMMITTED	Permitted	Permitted	Permitted
READ COMMITTED	--	Permitted	Permitted
REPEATABLE READ	--	--	Permitted
SERIALIZABLE	--	--	--

Table 1: ANSI isolation levels

Example queries

In these examples two transactions take place. In the first transaction, Query 1 is performed, then Query 2 is performed in the second transaction and the transaction committed, followed by Query 1 being performed again in the first transaction.

The queries use the following data table.

users

id	**name**	**age**
1	Joe	20
2	Jill	25

Repeatable reads (phantom reads)

A phantom read occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first. This can occur when range locks are not acquired on performing a SELECT.

Transaction 1 **Transaction 2**

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
                             /* Query 2 */
                             INSERT INTO users VALUES ( 3, 'Bob', 27 );
                             COMMIT;
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

Note that **transaction 1 executed the same query twice.** If the highest level of isolation were maintained, the same set of rows should be returned both times, and indeed that is what is mandated to occur in a database operating at the SQL SERIALIZABLE isolation level. However, at the lesser isolation levels, a different set of rows may be returned the second time.

In the SERIALIZABLE isolation mode, Query 1 would result in all records with age in the range 10 to 30 being locked, thus Query 2 would block until the first transaction was committed. In REPEATABLE READ mode, the range would not be locked, allowing the record to be inserted and the second execution of Query 1 to return the new row in its results.

Non-repeatable reads

In a lock-based concurrency control method, non-repeatable reads may occur when read locks are not acquired when performing a SELECT. Under multiversion concurrency control, non-repeatable reads may occur when the requirement that a transaction affected by a commit conflict must roll back is relaxed.

```
Transaction 1                     Transaction 2
/* Query 1 */
SELECT * FROM users WHERE id =
1;
                                  /* Query 2 */
                                  UPDATE users SET age = 21 WHERE id = 1;
                                  COMMIT; /* in multiversion concurrency
                                     control, or lock-based READ COMMITTED
                                  */
/* Query 1 */
SELECT * FROM users WHERE id =
1;
                                  COMMIT; /* lock-based REPEATABLE READ */
```

In this example, Transaction 2 commits successfully, which means that its changes to the row with id 1 should become visible. However, Transaction 1 has already seen a different value for *age* in that row. At the SERIALIZABLE and REPEATABLE READ isolation level, the DBMS must return the old value. At READ COMMITTED and READ UNCOMMITTED, the DBMS may return the updated value; this is a non-repeatable read.

There are two basic strategies used to prevent non-repeatable reads. The first is to delay the execution of Transaction 2 until Transaction 1 has committed or rolled back. This method is used when locking is used, and produces the serial schedule **T1, T2**. A serial schedule does not exhibit non-repeatable reads.

In the other strategy, which is used in **multiversion concurrency control**, Transaction 2 is permitted to commit first, which provides for better concurrency. However, Transaction 1, which commenced prior to Transaction 2, must continue to operate on a past version of the database — a snapshot of the moment it was started. When Transaction 1 eventually tries to commit, the DBMS looks to see if the result of committing Transaction 1 would be equivalent to the schedule **T1, T2**. If it is, then Transaction 1 can succeed. If it cannot be seen to be equivalent, however, Transaction 1 must roll back with a serialization failure.

Using a lock-based concurrency control method, at the REPEATABLE READ isolation mode, the row with ID = 1 would be locked, thus blocking Query 2 until the first transaction was committed or rolled back. In READ COMMITTED mode the second time Query 1 was executed the age would have changed.

Under multiversion concurrency control, at the SERIALIZABLE isolation level, both SELECT queries see a snapshot of the database taken at the start of Transaction 1. Therefore, they return the same data. However, if Transaction 1 were then to attempt to UPDATE that row as well, a serialization failure would occur and Transaction 1 would be forced to roll back.

At the READ COMMITTED isolation level, each query sees a snapshot of the database taken at the start of each query. Therefore, they each see different data for the updated row. No serialization failure is possible in this mode (because no promise of serializability is made) and Transaction 1 will not have to be retried.

READ UNCOMMITTED (dirty reads)

A dirty read occurs when a transaction reads data from a row that has been modified by another transaction, but not yet committed.

Transaction 1 **Transaction 2**

```
/* Query 1 */
SELECT * FROM users WHERE id =
1;
                                  /* Query 2 */
                                  UPDATE users SET age = 21 WHERE id =
                                  1;
                                  /* No commit here */
/* Query 1 */
SELECT * FROM users WHERE id =
1;
                                  COMMIT; /* lock-based DIRTY READ */
```