Introduction to Transaction Processing Concepts and Theory

Introduction

- **Transaction** provides a mechanism for describing logical units of database processing.
- A transaction is an atomic unit of database work (e.g., bank withdrawal) that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database.
- **Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions.
- These systems require high availability and fast response time for hundreds of concurrent users.

Introduction to Transaction Processing

- **Single-User System:** At most one user at a time can use the system.
- **Multiuser System**: Many users can access the system ***concurrently***.
- **Concurrency**
  - **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**: processes are concurrently executed in multiple CPUs.

Introduction to Transaction Processing

- **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.

Introduction to Transaction Processing

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database -** collection of named data items
- **Granularity of data** – size of data item like a field, a record , or a whole disk block
- Basic operations are **read** and **write**
  - **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.

Introduction to Transaction Processing

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer's main memory is <u>one block</u>. 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) command includes the following steps:**

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

Introduction to Transaction Processing

READ AND WRITE OPERATIONS (cont.):

- **write_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

FIGURE

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);
```

Introduction to Transaction Processing

Why do we need Concurrency Control?

If we do not control concurrency, we may encounter following three problems

- **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.

Introduction to Transaction Processing

Why do we need Concurrency Control?

- **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.

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem.

(a)

Time

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)

Some problems that occur when concurrent execution is uncontrolled. (b) The temporary update problem.

(b)

Time

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.

Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.

(c)

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*).

Introduction to Transaction Processing

Why recovery is needed:

(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.

Introduction to Transaction Processing

Why recovery is needed (cont.):

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.

   - should be programmed in the transaction itself.

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 .

Introduction to Transaction Processing

Why recovery is needed (cont.):

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 and System Concepts

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** - It is initial state. Transaction stays in this state while it is executing.
- **Partially committed state** - After the final statement has been executed, a transaction is in partially committed state.
- **Committed state** - After successful completion, a transaction is in committed state.
- **Failed state** - After the discovery that normal execution can no longer proceed, a transaction is in failed state.
- **Terminated State** – This state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later – either automatically or after being resubmitted by the user – as brand new transactions.

Transaction and System Concepts

Recovery manager keeps track of the following operations:

- **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 point 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.

Transaction and System Concepts

Recovery manager keeps track of the following operations (cont):

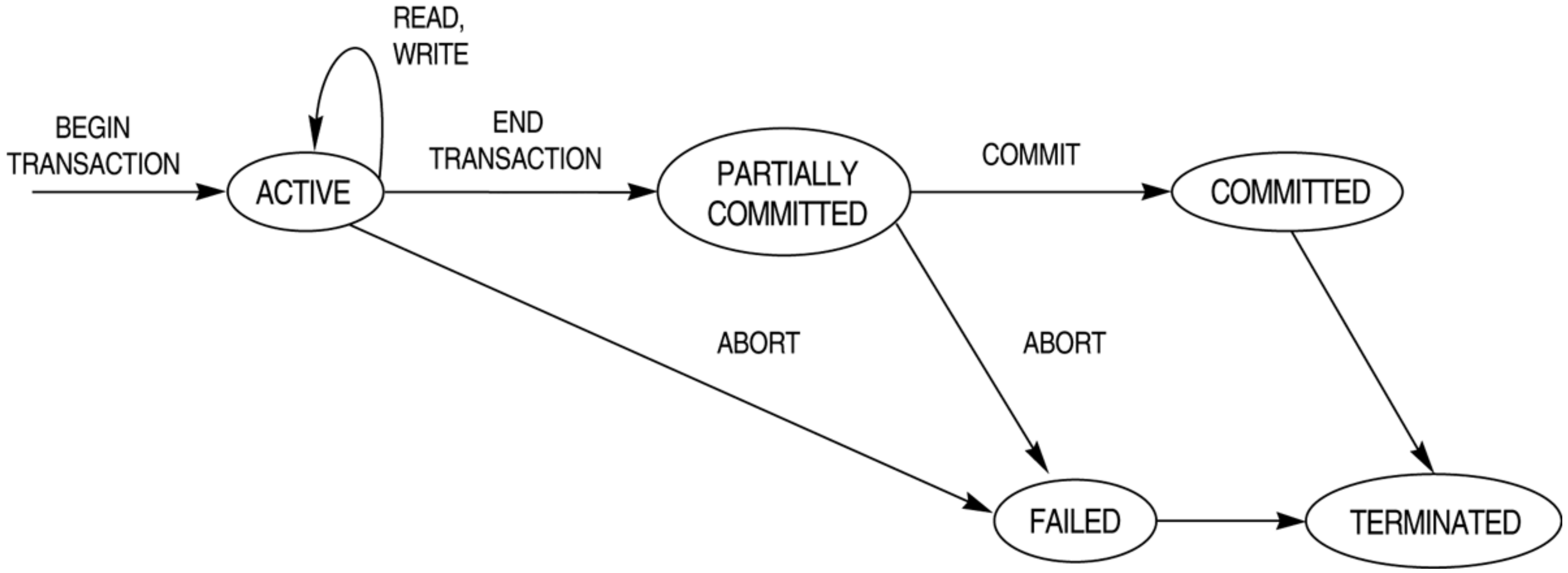
- **commit_transaction:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- **rollback (or abort):** This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be *undone.*

Transaction and System Concepts

Recovery techniques use the following operators:

- **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.

State transition diagram illustrating the states for transaction execution.



Transaction and System Concepts

The System Log

- **Log or Journal** : The log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from transaction failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.
- T in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

Transaction and System Concepts

The System Log (cont):

Types of log record:

1. [start_transaction,T]: Records that transaction T has started execution.
2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item,T,X]: Records that transaction T has read the value of database item X.
4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort,T]: Records that transaction T has been aborted.

Transaction and System Concepts

The System Log (cont):

- protocols for recovery that <u>avoid cascading rollbacks do not require that read operations be written to the system log</u>, whereas other protocols require these entries for recovery.
- strict protocols require simpler write entries that do not include new_value.

Transaction and System Concepts

Recovery using log records:

If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in later sections.

1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Transaction and System Concepts

Commit Point of a Transaction:

- **Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log. Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:** Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Transaction and System Concepts

Commit Point of a Transaction (cont):

- **Redoing transactions:** Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries. (Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost.)
- **Force writing a log:** *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

Desirable Properties of Transactions

ACID properties:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.

Desirable Properties of Transactions

ACID properties (cont.):

- **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.
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Characterizing Schedules based on Recoverability

- **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 Ti in S must appear in the same order in which they occur in Ti. Note, however, that operations from other transactions Tj <u>can be interleaved</u> with the operations of Ti in S.

Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

- **Recoverable schedule:** One where committed transaction need not to be rolled back.

  A schedule S is **recoverable** if transaction T in S does not commit 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.

Characterizing Schedules based on Recoverability

Schedules classified on recoverability (cont.):

- **Strict Schedules:** A schedule in which a transaction can neither **read** nor **write** an item X until the last transaction that wrote X has committed.

Characterizing Schedules based on Serializability

- **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.** Hence, in a serial schedule, only one transaction at a time is active-the commit (or abort) of the active transaction initiates execution of the next transaction.
- **Serializable schedule**: A schedule S (possibly concurrent) is **serializable** if it is **equivalent** to some serial schedule of the same n transactions.

Characterizing Schedules based on Serializability

- **Result equivalent**: Two schedules are called result equivalent if they produce the same final state of the database.
- **Conflict equivalent**: Two schedules are said to be conflict equivalent if the order of any two conflicting operations (**read** and **write**, **write** and **read**, and **write** and **write** on the same data item) is the same in both schedules.
- **Conflict serializable**: A schedule S is said to be conflict serializable if it is **conflict equivalent** to some serial schedule S'.

Characterizing Schedules based on Serializability

- Being serializable is <u>not</u> the same as being serial

- Being serializable implies that the schedule is a <u>correct</u> schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

Characterizing Schedules based on Serializability

- Serializability is hard to check.
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.

Characterizing Schedules based on Serializability

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends. Hence, we reduce the problem of checking the whole schedule to checking only a *committed project* of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
  - Use of locks with two phase locking

Characterizing Schedules based on Serializability

- **View equivalence**: A less restrictive definition of equivalence of schedules

- **View serializability:** definition of serializability based on view equivalence. A schedule is *view serializable* if it is ***view equivalent*** to a serial schedule.

Characterizing Schedules based on Serializability

Two schedules are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

Characterizing Schedules based on Serializability

The premise behind view equivalence:

- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results.
- **"The view"**: the read operations are said to see the *the same view* in both schedules.

Characterizing Schedules based on Serializability

Relationship between view and conflict equivalence:

- The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
- Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serialiable.
- **Any conflict serializable schedule is also view serializable, but not vice versa.**

Characterizing Schedules based on Serializability

Relationship between view and conflict equivalence (cont):

Consider the following schedule of three transactions

T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):

Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sa, the operations w2(X) and w3(X) are blind writes, since T2 and T3 do not read the value of X.

Sa is **view serializable**, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is **not conflict serializable**, since it is not conflict equivalent to **any serial schedule**.

Characterizing Schedules based on Serializability

Testing for conflict serializability

Algorithm:

1. Looks at only read_Item (X) and write_Item (X) operations
2. Constructs a precedence graph (serialization graph) - a graph with directed edges
3. An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
4. The schedule 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);	read_item (Z);	read_item (Y);
write_item (X);	read_item (Y);	read_item (Z);
read_item (Y);	write_item (Y);	write_item (Y);
write_item (Y);	read_item (X);	write_item (Z);
	write_item (X);	

FIGURE (continued)

Example of serializability testing. (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);	

Time ↓

Schedule E

FIGURE (continued)

Another example of serializability testing. Precedence graph for Schedule *E*.

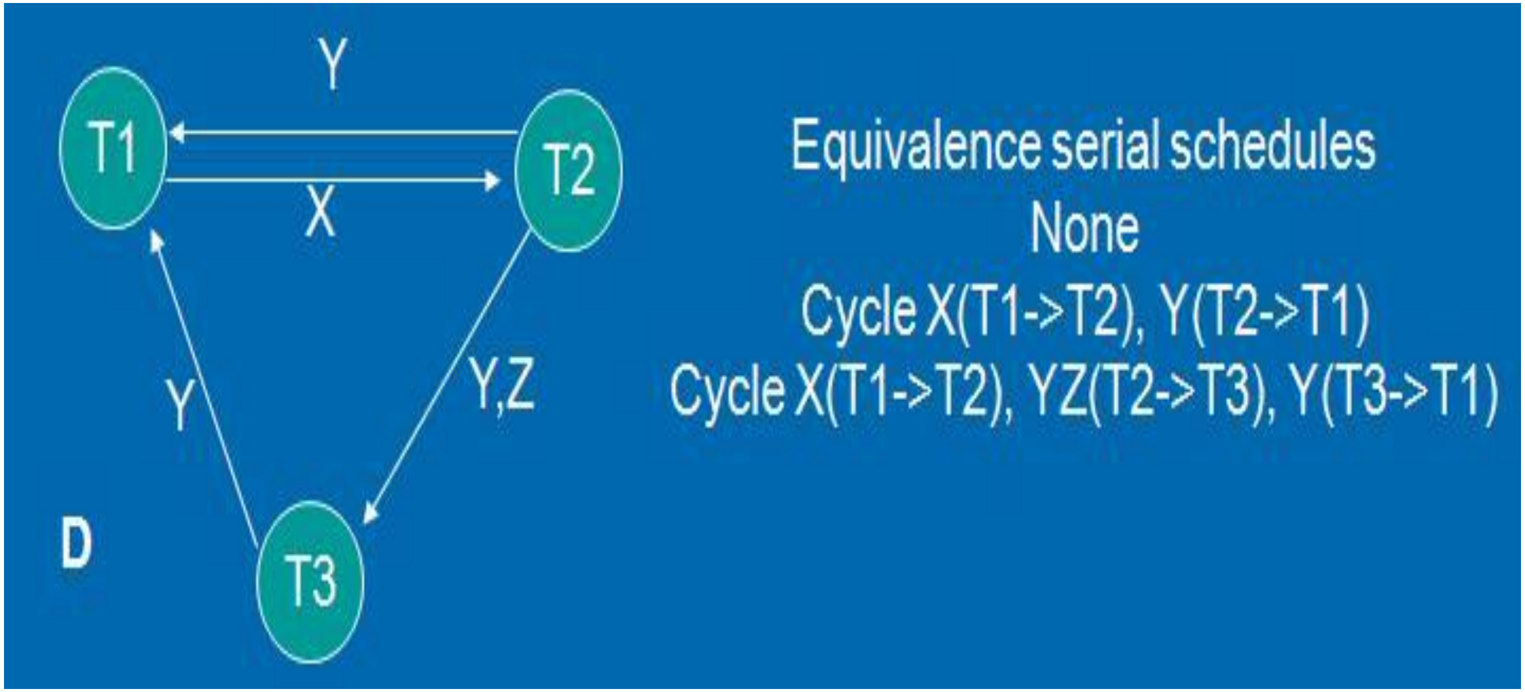


FIGURE (continued)

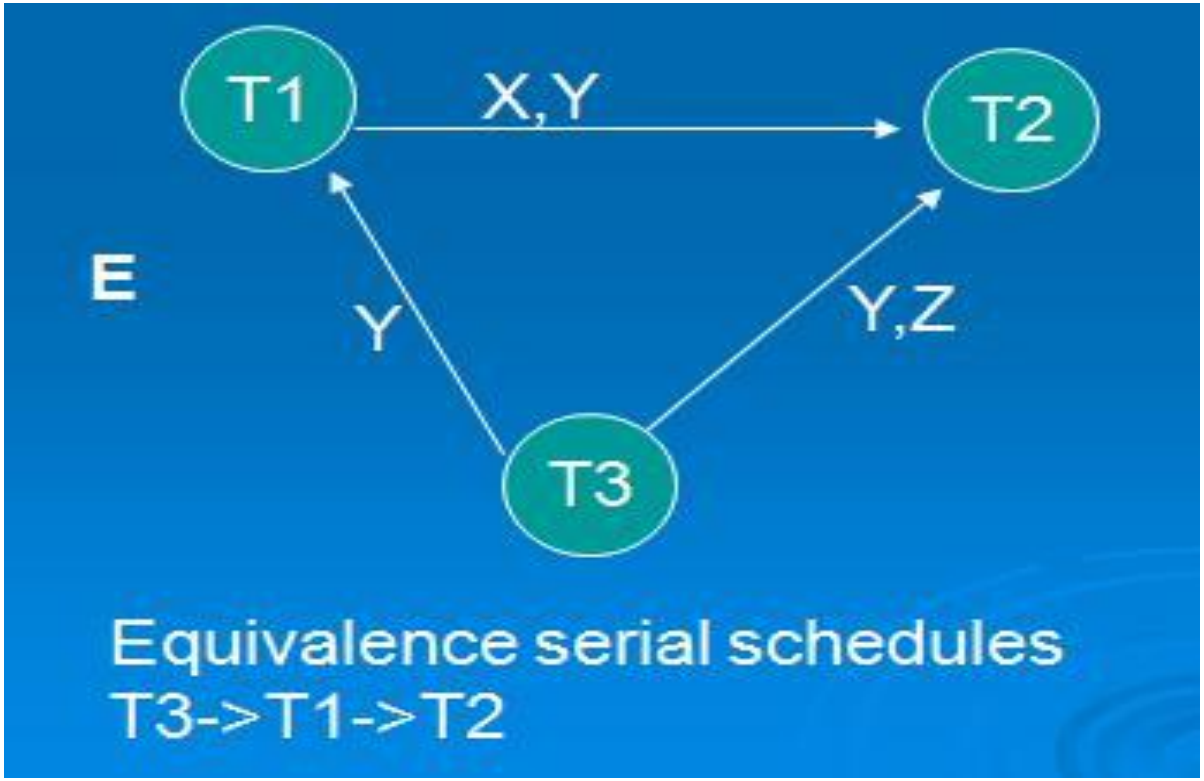
Example of serializability testing. (c) Schedule *F*.

(c)	transaction T_1	transaction T_2	transaction T_3
			read_item (Y);
			read_item (Z);
	read_item (X);		
Time ↓	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 (continued)

Another example of serializability testing. Precedence graph for Schedule F.



Characterizing Schedules based on Serializability

Other Types of Equivalence of Schedules

- Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly. Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

Characterizing Schedules based on Serializability

Other Types of Equivalence of Schedules(cont.)

Example: bank credit / debit transactions on a given item are **separable** and **commutative.**

Consider the following schedule S for the two transactions:

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

Using conflict serializability, it is not **serializable.**

However, if it came from a (read,update, write) sequence as follows:

r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20;w2(Y);r1(Y);

Y := Y + 10; w1(Y); r2(X); X := X + 20; w2(X);

Sequence explanation: debit, debit, credit, credit.

It is a **correct schedule** **<u>for the given semantics</u>**