CSE2004 – Database Management Systems

Text Books :

1.R. Elmasri & S. B. Navathe, Fundamentals of Database Systems, Addison Wesley, 7 th Edition, 2015

2.Raghu Ramakrishnan, Database Management Systems,Mcgraw-Hill,4th edition,2015

Reference Book

3.A. Silberschatz, H. F. Korth & S. Sudershan, Database System Concepts, McGraw Hill, 6th Edition 2010

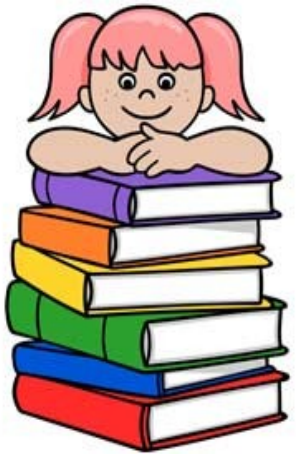
4.Thomas Connolly, Carolyn Begg," Database Systems : A Practical Approach to Design, Implementation and Management",6th Edition,2012

5.Shashank Tiwari ,"Professional NoSql",Wiley ,2011

VIT UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
www.vit.ac.in
Vellore ▪ Chennai
CHENNAI CAMPUS
Vandalur - Kelambakkam Road, Chennai - 600127

Unit – 4 : QUERY PROCESSING AND TRANSACTION PROCESSING

- ✓ Translating SQL Queries into Relational Algebra
- ✓ Heuristic query optimization
- ✓ Introduction to Transaction Processing
- ✓ Transaction and System concepts
- ✓ Desirable properties of Transactions
- ✓ Characterizing schedules based on recoverability
- ✓ Characterizing schedules based on serializability



VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
www.vit.ac.in
Vellore ▪ Chennai
CHENNAI CAMPUS
Vandalur - Kelambakkam Road, Chennai - 600127

Introduction to Transaction Processing

Introduction to Transaction Processing

Single-User System: At most one user at a time can use the system.

Multi-user 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 HLL (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.

SIMPLE MODEL OF A DATABASE

A database is a collection of named data items

Granularity : Size of data Item

– 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.

READ and WRITE OPERATIONS:

– Basic unit of data transfer from the disk to the computer main memory is one block.

In general, a data item (Read / 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:

- 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.

READ and WRITE OPERATIONS:

– Basic unit of data transfer from the disk to the computer main memory is one block.

In general, a data item (Read / 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.

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) 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);
```

Why Concurrency Control is needed ?

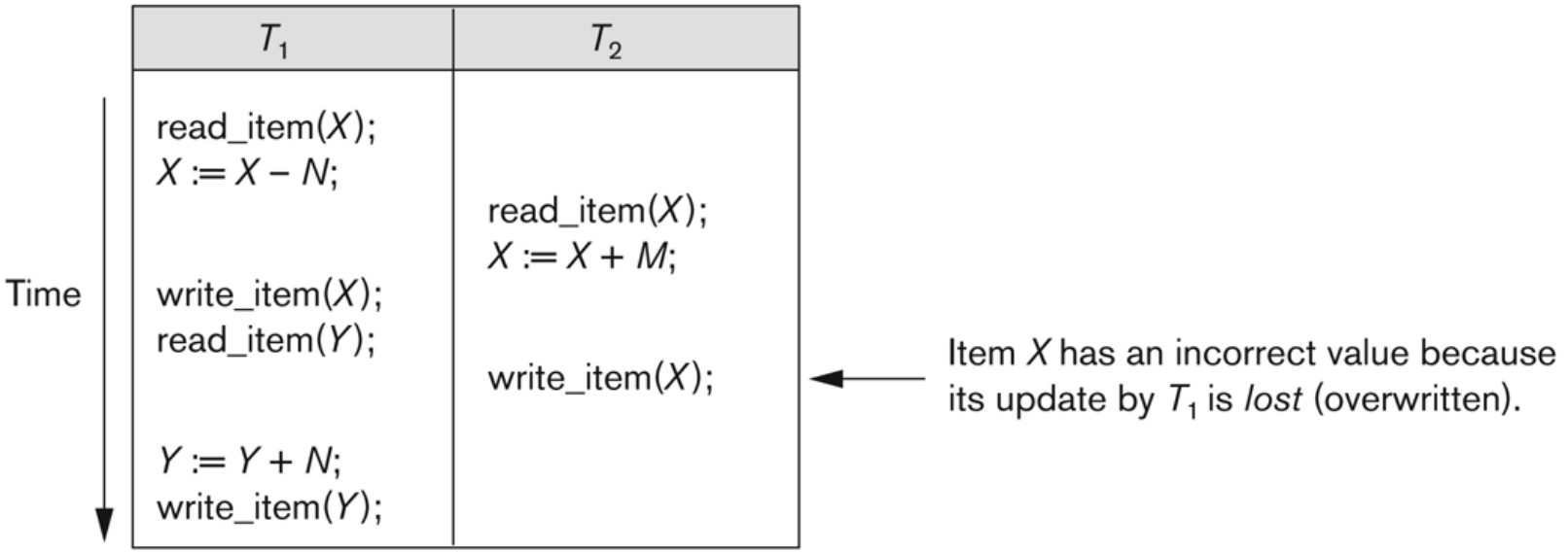
The following are the reasons

- 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.

Concurrency Control is Uncontrolled

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(a)

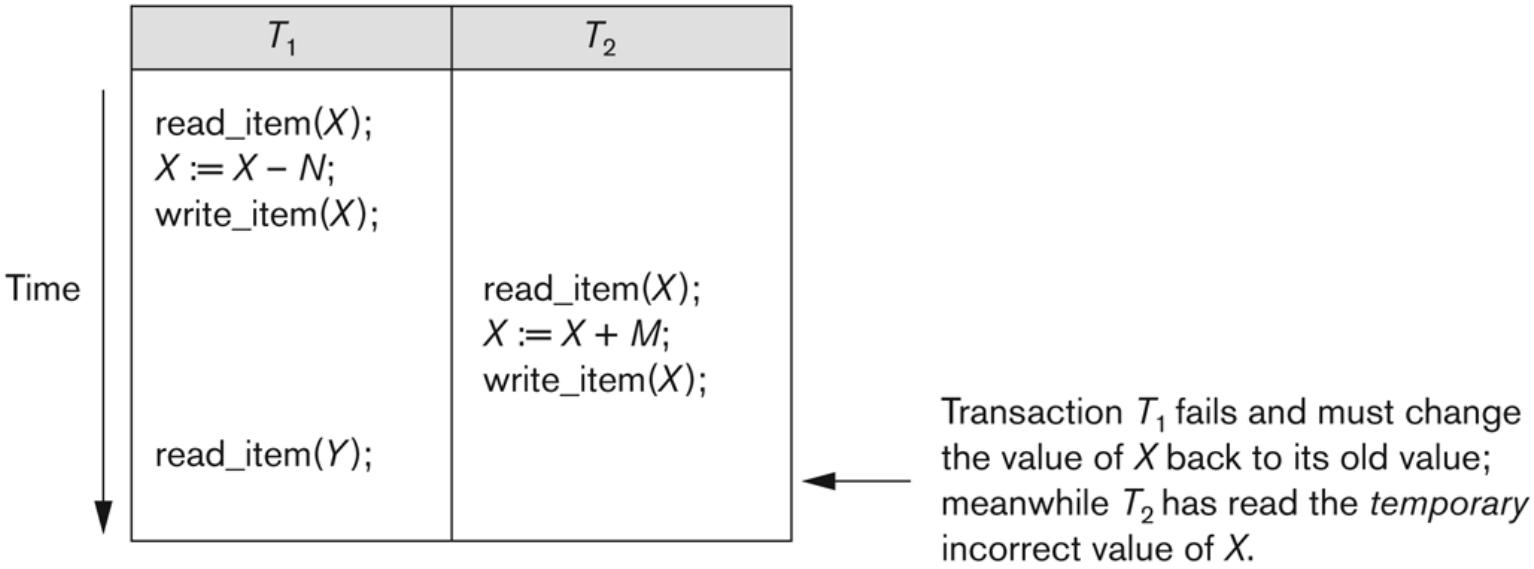


Lost Update Problem

Concurrency Control is Uncontrolled

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(b)



Temporary Update Problem

Concurrency Control is Uncontrolled

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

Incorrect Summary Problem

Transaction Recovery

Why recovery is needed ? (What causes a Transaction to fail)

- 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.

- A transaction/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.

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

Transaction Recovery

Why recovery is needed ? (What causes a Transaction to fail)

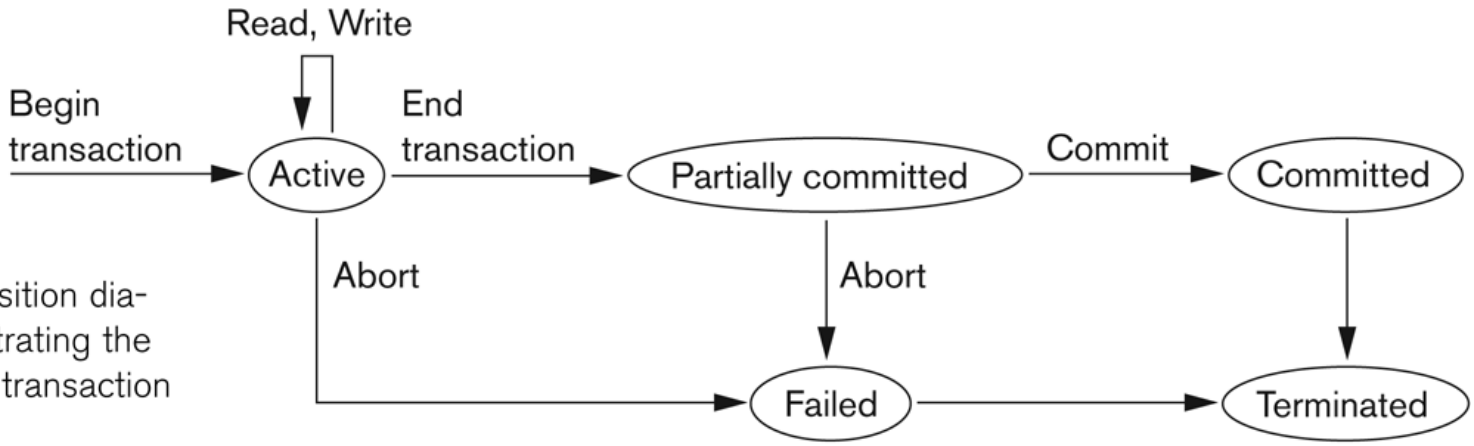
- Concurrency control enforcement: The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock
- Disk failure: Some disk blocks may lose their data because of a read or write 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.

Transactions and System Concepts

- A **transaction** is an atomic unit of work that is either completed in its entirely or not done at all(Commit/Rollback).
- For Recovery, the system needs to keep track of when the transaction starts, terminates, and commits or aborts (Recovery Manager).

Transaction states:

- ✓ Active state
- ✓ Partially committed state
- ✓ Committed state
- ✓ Failed state
- ✓ Terminated State



State transition diagram illustrating the states for transaction execution.

Transactions 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 limit of transaction execution.

At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied or 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.

Recovery manager keeps track of the following operations:

- **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.

The System 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.

Transactions and System Concepts

For Eg., Consider the Transaction 'T' refers to a unique **transaction-id,** generated automatically by the system and is used to identify each transaction:

Types of log record:

- [start_transaction,T]: Records that transaction T has started execution.
- [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.
- [read_item,T,X]: Records that transaction T has read the value of database item X.
- [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- [abort,T]: Records that transaction T has been aborted.

Commit Point of a Transaction:

Definition a Commit Point:

- 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.
- The transaction ' T ' 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.

Commit Point of a Transaction:

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 Transaction

ACID properties:

- **Atomicity(A):** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation(C)**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation(I)**: 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(D)**: 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 is known as a transaction schedule (or history).

A **schedule** (or **history**) S(n) transactions T_1, T_2, …, T_n:

It 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_1 in S must appear in the same order in which they occur in T_1.

Note : However, that operations from other transactions T_j can be interleaved with the operations of T_i in S.

Characterizing Schedules based on Recoverability

Schedules classification:

Recoverable schedule:

- Where no transaction needs to be rolled back.
- A schedule S is recoverable, if no transaction T in S commits until all transactions T' have written an item, that T reads have committed.

Cascadeless schedule:

- 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/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.
- Serializable schedule:
  A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
- 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 is the same in both schedules.

Example: Interleaved Transactions

Consider two transactions:

```
T1: BEGIN   A=A+100,   B=B-100   END
T2: BEGIN   A=1.06*A,   B=1.06*B   END
```

One possible interleaved execution:

```
T1: A=A+100,                    B=B-100
T2:              A=1.06*A,              B=1.06*B
```

It is OK. But what about another interleaving?

```
T1: A=A+100,                              B=B-100
T2:              A=1.06*A, B=1.06*B
```

Schedule: Modeling Concurrency

- *Schedule*: a sequence of operations from a set of transactions, where operations from any one transaction are in their original order

Notation:

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

$R_i(X)$: read X by T_i

$W_i(X)$: write X by T_i

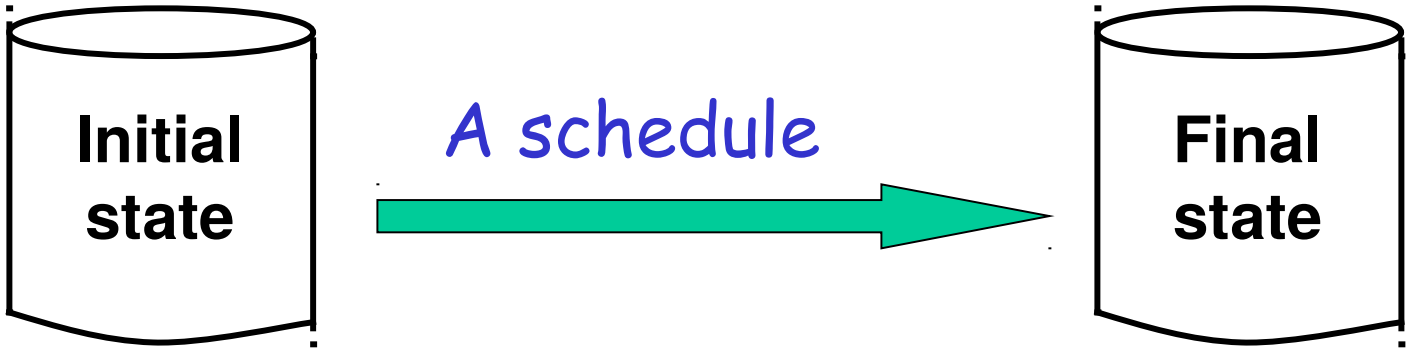
$R_1(A), W_1(A), R_2(B), W_2(B), R_1(C), W_1(C)$

Schedule (cont.)

Represents some actual sequence of database actions.

In a complete schedule, each transaction ends in commit or abort.

A schedule transforms database from an initial state to a final state



Schedule (cont.)

- Assume a consistent initial state
- A representation of an execution of operations from a set of transactions
- Ignore
  - ✔ aborted transactions
  - ✔ Incomplete (not yet committed) transactions
- Operations in a schedule conflict if
  1. They belong to different transactions
  2. They access the same data item
  3. At least one item is a write operation

Anomalies with Concurrency

Interleaving transactions may cause many kinds of consistency problems

Reading Uncommitted Data (“dirty reads”):

$$R_1(A), W_1(A), R_2(A), W_2(A), C_2, R_1(B), A_1$$

- Unrepeatable Reads:

$$R_1(A), R_2(A), W_2(A), C_2, R_1(A), W_1(A), C_1$$

- Overwriting Uncommitted Data (lost update):

$$R_1(A), R_2(A), W_2(A), W_1(A)$$

Anomalies with Concurrency

Incorrect Summary Problem

Data items may be changed by one transaction while another transaction is in the process of calculating an aggregate value

A correct “sum” may be obtained prior to any change, or immediately after any change

Serial Schedule

An acceptable schedule must transform database from a consistent state to another consistent state

- *Serial schedule* : one transaction runs entirely before the next transaction starts.

$R_1(X)\ W_1(X)\ C_1\ R_2(X)\ W_2(X)\ C_2$

$R_2(X)\ W_2(X)\ C_2\ R_1(X)\ W_1(X)\ C_1$

} Serial

$R_1(X)\ R_2(X)\ W_2(X)\ W_1(X)\ C_1\ C_2$

Non-serial

Serial Schedule IS Acceptable

Serial schedules guarantee transaction isolation & consistency

Different serial schedules can have different final states

N transactions may form N! different serial schedules

Any state from a serial schedule is acceptable – DBMS makes no guarantee about the order in which transactions are executed

Example: Serial Schedules

T1: R(X), X=X+10, W(X)

T2: R(X), X=X*2, W(X)

Initial
X = 20

S1: $R_1(X)\ W_1(X)\ C_1\ R_2(X)\ W_2(X)\ C_2$

Final
X = 60

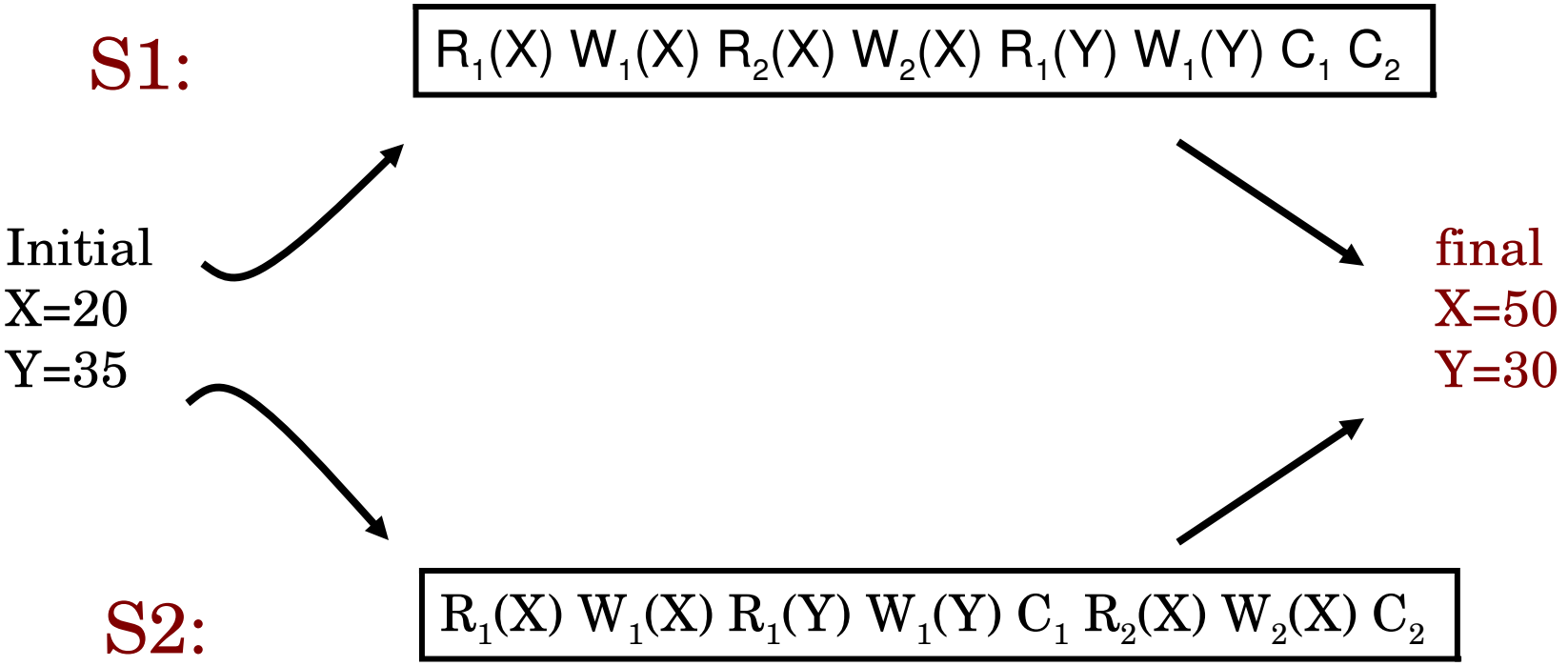
S2: $R_2(X)\ W_2(X)\ C_2\ R_1(X)\ W_1(X)\ C_1$

Final
X = 50

Is Non-Serial Schedule Acceptable?

T1: R(X), X=X*2, W(X), R(Y), Y=Y-5, W(Y)

T2: R(X), X=X+10, W(X)



Serializable Schedules

- *Serializable schedule*: Equivalent to a serial schedule of committed transactions.

Non-serial (allow concurrent execution)

Acceptable (final state is what some serial schedule would have produced)

Types of Serializable schedules: depend on how the equivalency is defined

Conflict: based on conflict operations

View: based on viewing of data

Characterizing Schedules based on Serializability

- Conflict serializable:
  A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.
- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct 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.
- 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 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

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 iff :

- The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
- 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 (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 $R_i(X)$ of T_i in S'.
- 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 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 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 serializable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

Characterizing Schedules based on Serializability

Relationship between view and conflict equivalence:

- 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 T1 and T3 do not read the value of X.

Here, 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

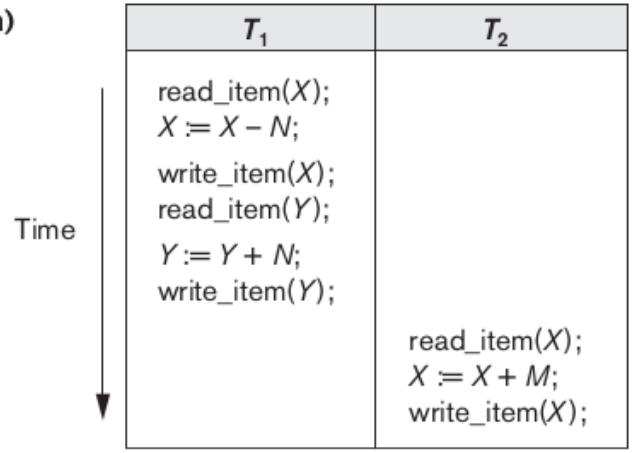
Testing for conflict serializability:

Algorithm:

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

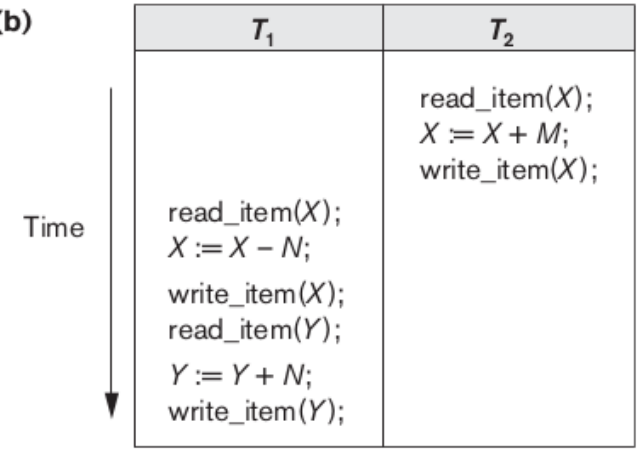
Characterizing Schedules based on Serializability

(a)



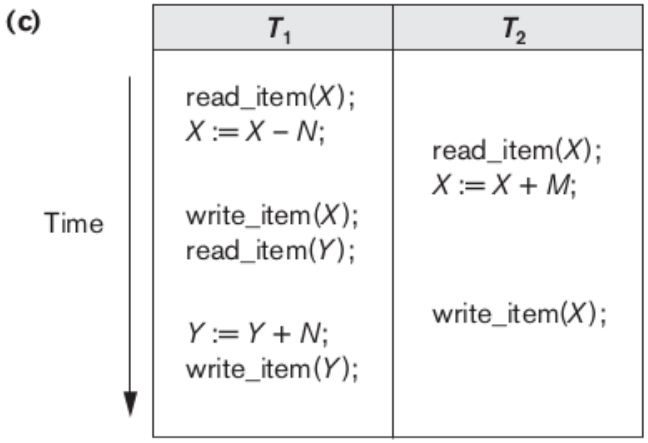
Schedule A

(b)

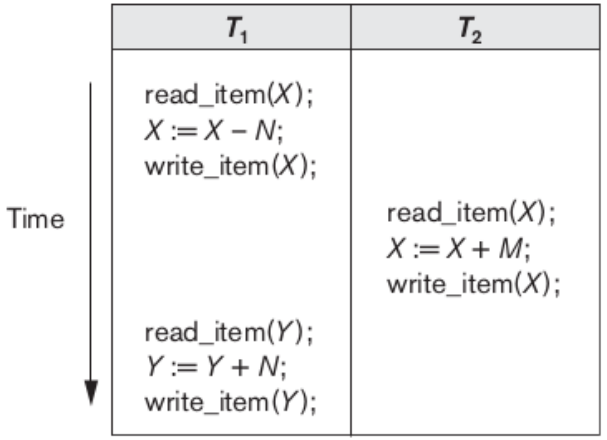


Schedule B

(c)



Schedule C

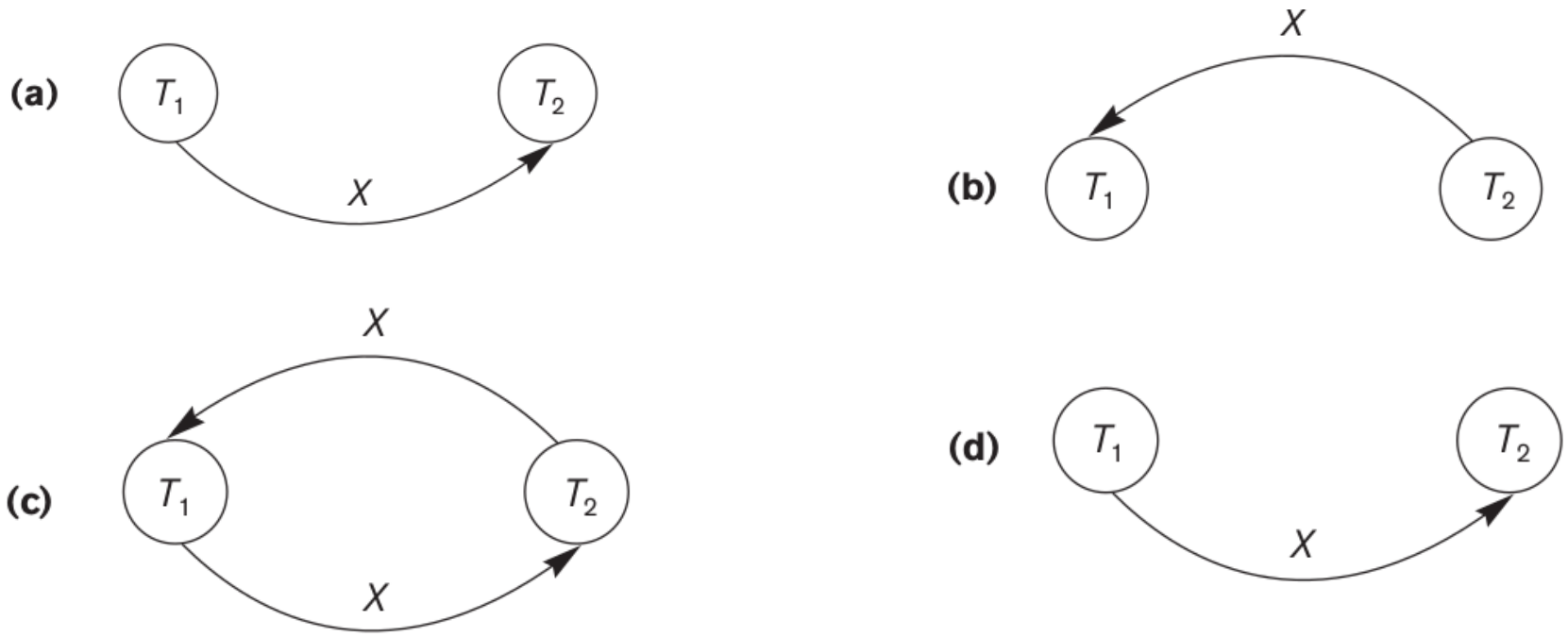


Schedule D

Characterizing Schedules based on Serializability

FIGURE : Constructing the precedence graphs for schedules A,B,C & D to test for conflict serializability.

(a) Precedence graph for serial schedule A.
(b) Precedence graph for serial schedule B.
(c) Precedence graph for schedule C (not serializable).
(d) Precedence graph for schedule D (serializable, equivalent to schedule A).



Characterizing Schedules based on Serializability

Example-1

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

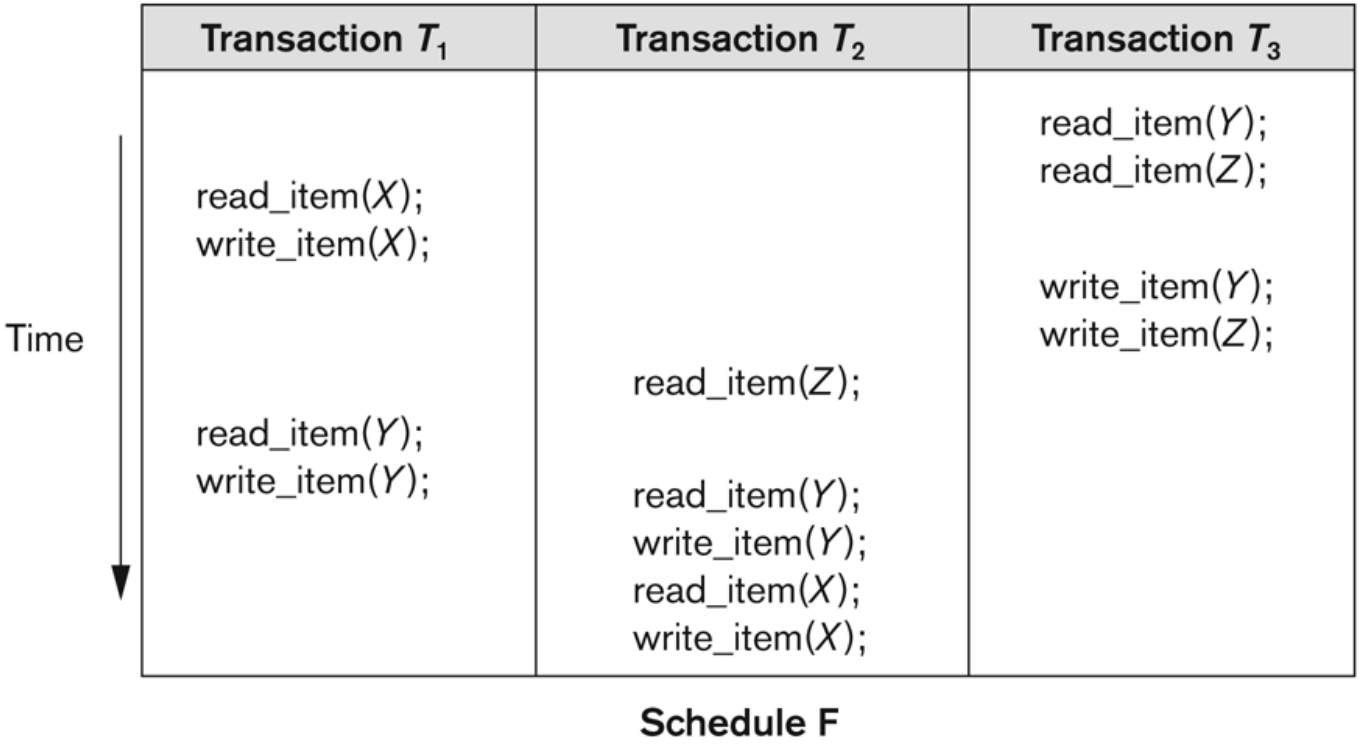
Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

Time

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

Characterizing Schedules based on Serializability



Schedule F

Constructing the precedence graphs for schedules E and F to test for conflict serializability.


Queries