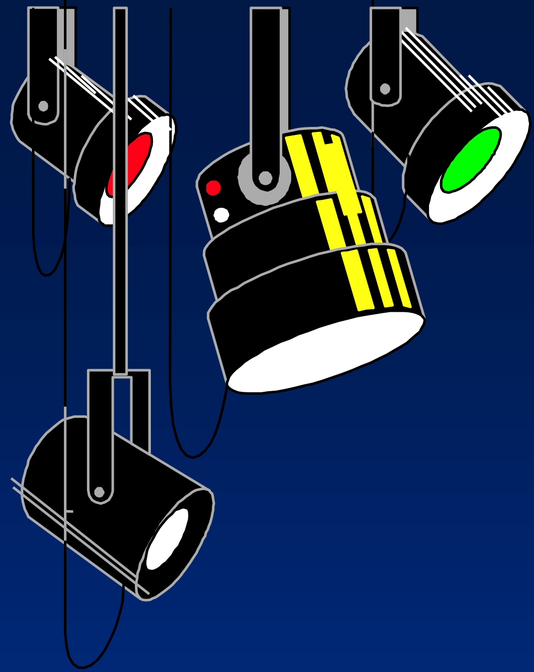
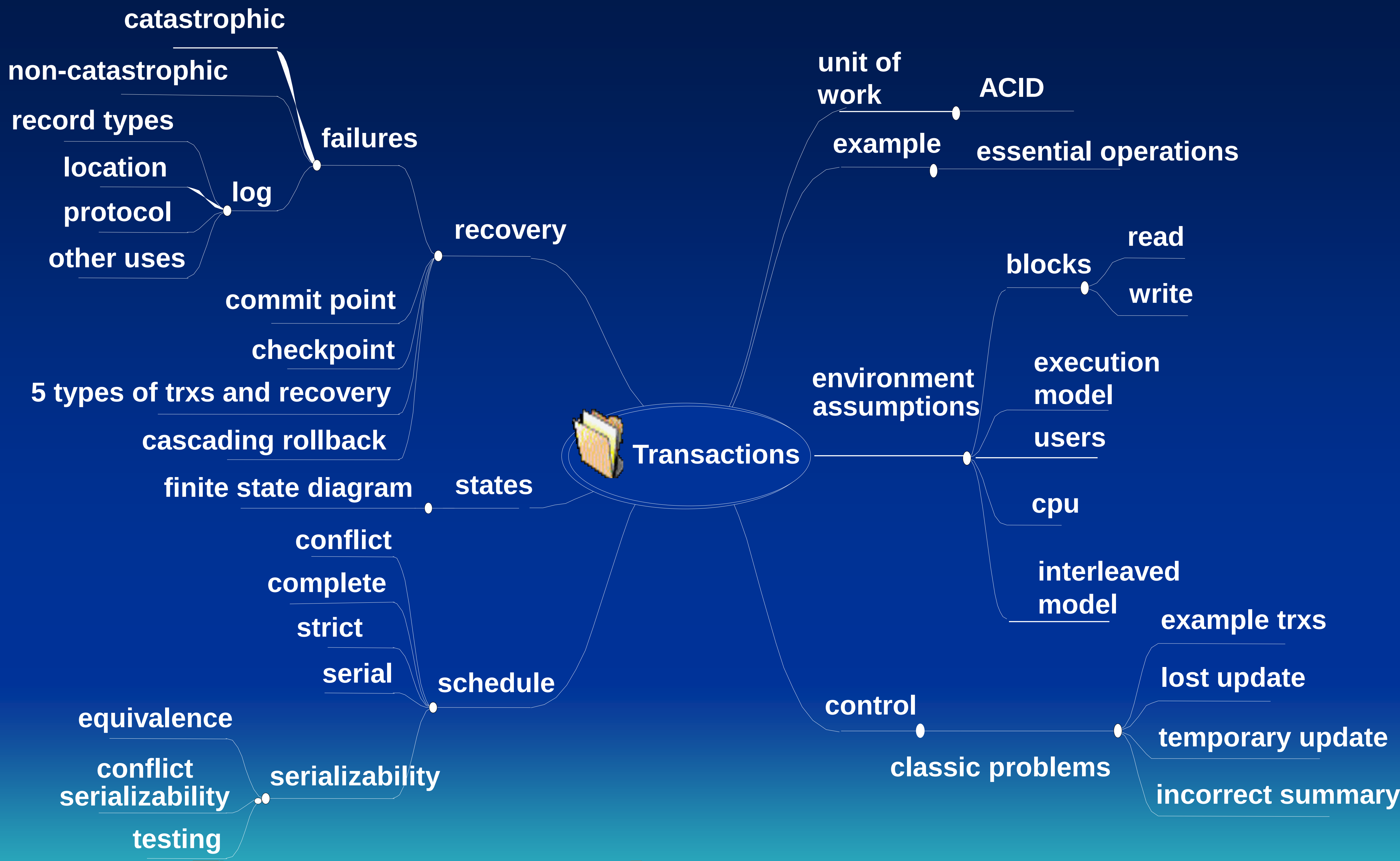


Transaction processing concepts

(Ch. 19, 3rd ed. – Ch. 17, 4th ed., 5th ed. – Ch. 21, 6th ed. – Ch. 20, 7th ed.)


Transactions
recovery
failures
catastrophic
non-catastrophic
log
record types
location
protocol
other uses
commit point
checkpoint
5 types of trxs and recovery
cascading rollback
states
finite state diagram
schedule
conflict
complete
strict
serial
serializability
equivalence
conflict serializability
testing
unit of work
ACID
example
essential operations
environment assumptions
blocks
read
write
execution model
users
cpu
interleaved model
control
classic problems
example trxs
lost update
temporary update
incorrect summary

Transaction Processing

From a **technical perspective**, a transaction is a unit of work in a database system

From a **user's perspective**, a program execution that accomplishes a useful task, such as:

- Change someone's name
- Change someone's address
- Withdraw money from an account
- Transfer money from one account to another account
- Reserve a seat on a flight
- etc

In an online transaction processing (**OLTP**) environment, a transaction is usually small and fast. It is something that must get through the system quickly to give (typically) sub-second response.

To generate faith in the computing system, a transaction will have the **ACID** properties:

- Atomic – a transaction is done in its entirety, or not at all
- Consistent – a transaction leaves the database in a correct state. This is generally up to the programmer to guarantee.
- Isolation – a transaction is isolated from other transactions so that there is not adverse inter-transaction interference
- Durable – once completed (committed) the result of the transaction is not lost.

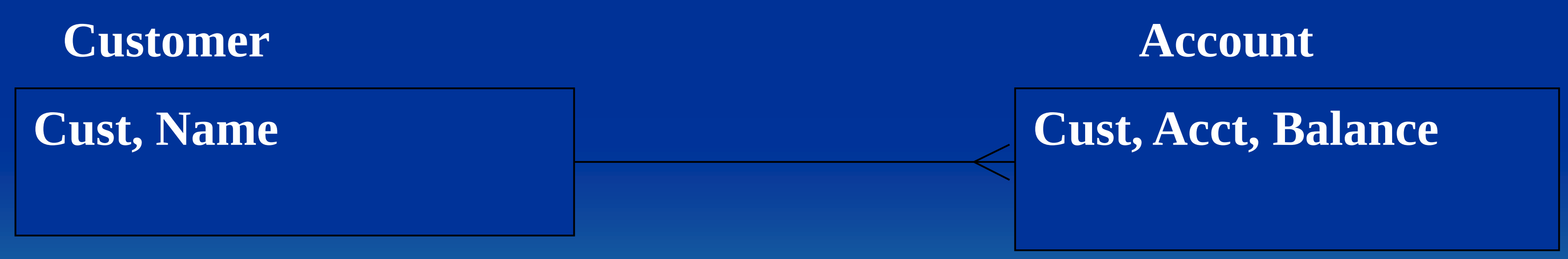
Example of a transaction coded in a 3GL

Consider a transaction that transfers money between a customer's accounts.

This example uses the C programming language with embedded SQL.

Note that each execution of the program would be a new transaction.

Database:



Code Fragment

…… definitions of program data areas

cust_no
from_account
to_account
trans_amount
cust_name

```
Printf("\nEnter customer identifier:");
Scanf("%s", cust_no);
```

```
EXEC SQL set transaction
            read write;
```
Begin transaction

```
EXEC SQL
    Select Name into :cust_name
    From Customer
    Where Cust=:cust_no;
```
Read (Customer.cust_name)

Code Fragment continued

```
Printf("\nHello(%s)", cust_name);

Printf("\nTransfer from account:");
Scanf("%s", from_account);

Printf("\nTransfer to account:");
Scanf("%s", to_account);

Printf("\nTransfer the amount:");
Scanf("%s", trans_amount);
```

Code Fragment continued

```
EXEC SQL                              Write (Account.to_acct.Balance)
        Update Account
        Set Balance = Balance + :trans_amount
            Where Cust = :cust_no
            And Acct = :to_acct;

EXEC SQL                              Write (Account.from_acct.Balance)
        Update Account
        Set Balance = Balance - :trans_amount
            Where Cust = :cust_no
            And Acct = :from_acct;

EXEC SQL                              Commit
        Commit transaction;
```

Environment

Blocks

- Data is stored in blocks on disk.

- The layout of blocks is controlled by the system. You may have a choice of variable or fixed length blocks and of a specific maximum blocksize (although the dba group may have chosen to always use one or two blocksizes (maybe 4K and 32K) to simplify the system).

- Typicaly there are several records per block which has the effect of
  - Increasing storage utilization, and
  - Decreasing the number of transfers required between memory and disk.

Environment

What happens when READ(X) is executed?

- The DBMS determines the address of the block holding X
- The block is transmitted from disk to a buffer
- X is copied from the buffer to a program variable

***Read* (*Customer.cust_name*)**

```
EXEC SQL
    Select Name into :cust_name
    From Customer
    Where Cust=:cust_no;
```

Environment

What happens when WRITE(X) is executed?

- The DBMS determines the address of the block holding X
- Unless the block is already in a buffer, the block is transmitted from disk to a buffer
- X (new value for X) is copied from program variables into the buffer
- The buffer is written out to disk (a delay may occur here)

```
EXEC SQL                                  Write (Account.to_acct.Balance)
    Update Account
    Set Balance = Balance + :trans_amount
        Where Cust = :cust_no
        And Acct = :to_acct;
```

Environment

Users
Multiple users access the database at the same time

Program Execution Model
Multiple programs are executed concurrently.

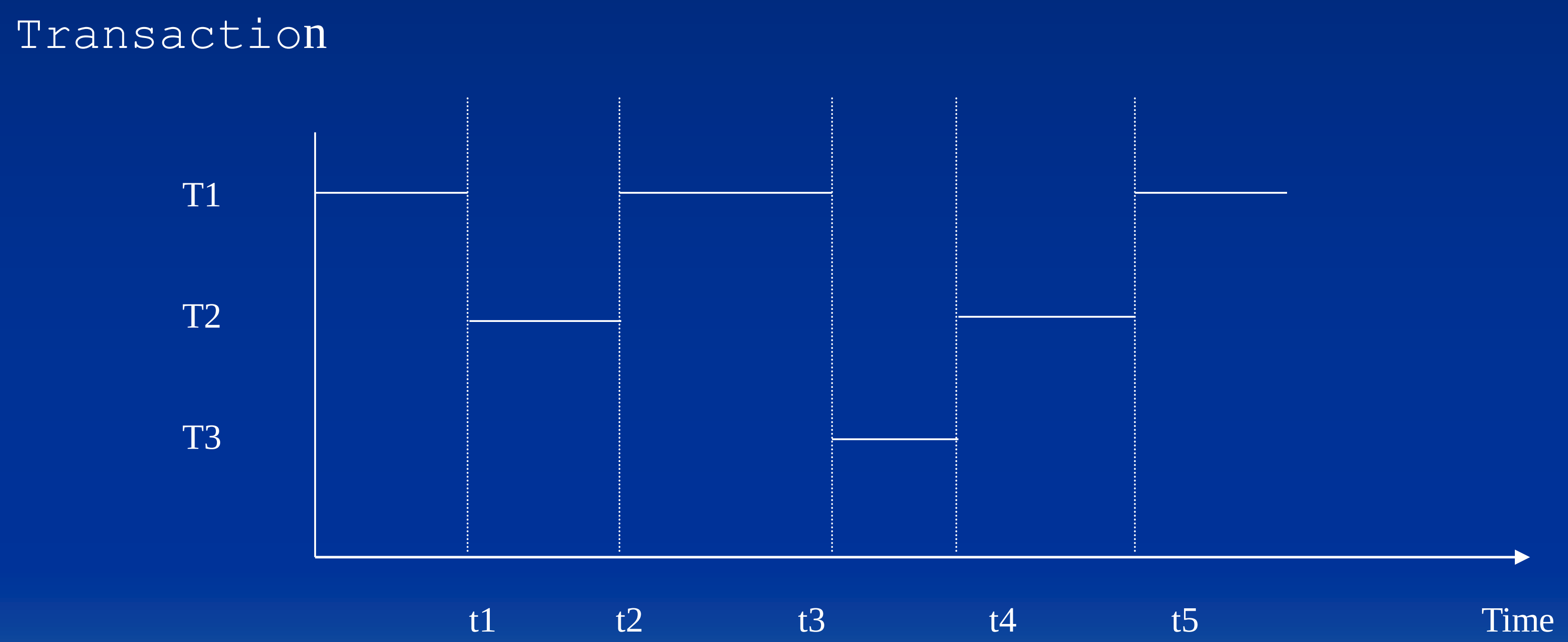
Processor Model
A single processor. The theory developed for transaction concurrency is based on a single processor and can be adapted for multiple processor situations.

These last three assumptions lead us to the **Interleaved** model of transaction execution.

Environment

Interleaved model of transaction execution

Several transactions, initiated by any number of users, are concurrently executing. Over a long enough time interval, several transactions may have executed without any of them completing.



Why do transactions need Concurrency Control?

If we do not protect transactions from other transactions the database can become inconsistent and/or incorrect information derived from the database.

Consider the 3 classic transaction problems

- Lost update
- Temporary update
- Incorrect summary

Why do transactions need Concurrency Control?

Our examples will deal with a simple database intended to keep track of the number of seats reserved on individual flights.

Flight

F-id, F_seats_res, …

Sample data

F-id	F_seats_res
10	120
12	160
20	100
……	

At any given point in time any number of users may be entering transactions into the system.

Suppose we have three types of transactions:

- Cancel N seats on one flight and reserve N seats on another
- Reserve M seats on a flight
- Count the total number of reservations

Why do transactions need Concurrency Control?

We outline the transactions below.
For simplicity we make a liberal interpretation of our statements.
For example:

- READ(X): read the flight record for flight X. (To simplify our notation, we assume that the program variable is also named X.)
- X:=X - N: F_seats_res for flight X is decremented by N.
- X:=X + M: F_seats_res for flight X is increased by M.
- WRITE(X): write the value of program variable X into F_seats_res for flight X.

Why do transactions need Concurrency Control?

Transaction1	Transaction2	Transaction3
READ(X)	READ(X)	SUM:=0
X:=X-N	X:=X+M	READ(X)
WRITE(X)	WRITE(X)	SUM:=SUM+X
READ(Y)		READ(Y)
Y:=Y+N		SUM:=SUM+Y
WRITE(Y)		READ(Z)
		SUM:=SUM+Z

Transaction1:

Exec SQL select F_seats_res into X'
from Flights
where F-id = X

X' := X' – N

Exec SQL update Flights
Set F_seats_res = X'
where F-id = X

⋮
⋮

Lost Update Problem

We have Transactions 1 and 2 concurrently executing in the system. They happen to interleave in the following way, which results in an incorrect value stored for flight X (try this for X=10, Y=12, N=5 and M=8).

Time	Transaction1	Transaction2
1	READ(X)	
2	X:=X-N	
3		READ(X)
4		X:=X+M
5	WRITE(X)	
6	READ(Y)	
7		WRITE(X)
8	Y:=Y+N	
9	WRITE(Y)	

Time	Transaction1	Transaction2
1	p1_X = 10	
2	p1_X = 5	
3		p2_X = 10
4		p2_X = 18
5	d_X = 5	
6	p1_Y = 12	
7		d_X = 18
8	p1_Y = 17	
9	d_Y = 17	

P1

N = 5;
X;
Y;
… …

P2

M = 8;
X;
Y;

… …

F-id	F-seats-res.
X	10
Y	12
… …	

X = 10
Y = 12
N = 5
M = 8

Transaction1	Transaction2
READ(X)	READ(X)
X:=X-N	X:=X+M
WRITE(X)	WRITE(X)
READ(Y)	
Y:=Y+N	
WRITE(Y)	

T1 → T2

d_X = 5

d_Y = 17

d_X = 13

d_X and d_Y represent th values of X and Y in the database.

T2 → T1

d_X = 18

d_X = 13

d_Y = 17

P1

```
N = 5;
X;
Y;
… …
```

P2

```
M = 8
X;
Y;

… …
```

F-id	F-seats-res.
X	10
Y	12
… …	

F-id	F-seats-res.
X	10
Y	12
… …	

by executing P1 and P2 serially →

F-id	F-seats-res.
X	13
Y	17
… …	

Why do transactions need Concurrency Control?

Temporary Update Problem

We have transactions 1 and 2 running again. This time Transaction 1 terminates before it completes – it just stops, perhaps it tried to execute an illegal instruction or accessed memory outside its allocation. The important point is that it doesn't complete its unit of work; Transaction 2 reads 'dirty data' using a value derived from an inconsistent database state.

Time	Transaction1	Transaction2
1	READ(X)	
2	X:=X-N	
3	WRITE(X)	
4		READ(X)
5		X:=X+M
6		WRITE(X)
7	READ(Y)	
8	terminates!	

Transaction2 reads a 'dirty' value – one that Transaction1 has not committed to the database

X should be rolled back to what it was at Time2

Why do transactions need Concurrency Control?

Incorrect Summary Problem

Transactions 1 and 3 are executing and interleaved in such a way that the total number of seats calculated by transaction 3 is incorrect.
(X=10, Y=12, Z = 2, N=5 and M=8)

Time	Transaction1	Transaction3	Time	Transaction1	Transaction3
1		SUM:=0	1		SUM = 0
2	READ(X)		2	p1_X =10	
3	X:=X-N		3	p1_X = 5	
4	WRITE(X)		4	d_X = 5	
5		READ(X)	5		p3_X = 5
6		SUM:=SUM+X	6		SUM = 5
7		READ(Y)	7		p3_Y = 12
8		SUM:=SUM+Y	8		SUM = 17
9	READ(Y)		9	p1_Y = 12	
10	Y:=Y+N		10	p1_Y = 17	
11	WRITE(Y)		11	d_Y = 17	
12		READ(Z)	12		p3_Z = 2
13		SUM:=SUM+Z	13		SUM = 19

X = 10
Y = 12
Z = 2
N = 5
M = 8

Transaction1	Transaction3
READ(X)	SUM:=0
X:=X-N	READ(X)
WRITE(X)	SUM:=SUM+X
READ(Y)	READ(Y)
Y:=Y+N	SUM:=SUM+Y
WRITE(Y)	READ(Z)
	SUM:=SUM+Z

T1 → T3

d_X = 5

d_Y = 17

d_Z = 2

sum = 24

T3 → T1

sum = 24

d_X = 5

d_Y = 17

d_Z = 2

Why do we need to provide transaction recovery?

Transactions can fail:

- Catastrophic (media failure)
  Hard disk crash
  Fire, theft, flood, …

- Non-catastrophic (system failure)
  Computer failure – memory becomes unreliable
  Transaction error – e.g. divide by zero
  Transaction aborts itself
  Concurrency control system aborts a transaction

To allow for recovery we use a **Log,** which contains several records for each transaction

1. [start_transaction, T] Indicates that transaction T has started execution.
2. [write_item, T, X, old_value, new_value] Indicates that transaction T has changed the value of database item X from old_value to new_value.
3. [Read_item, T, X] Indicates that transaction T has read the value of database item X.
4. [commit, T] Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort, T] Indicates that transaction T has been aborted.
6. [Checkpoint]: A checkpoint record is written into the log periodically. At that point, the system writes out to the database on disk all DBMS buffers that have been modified.

Example: [write_item, 1, (account, ‘123456789’, balance), 10,000, 13,000]

To allow for recovery we use a **Log**

- The log should be on a separate disk
- The system always writes to the log before it writes to the database

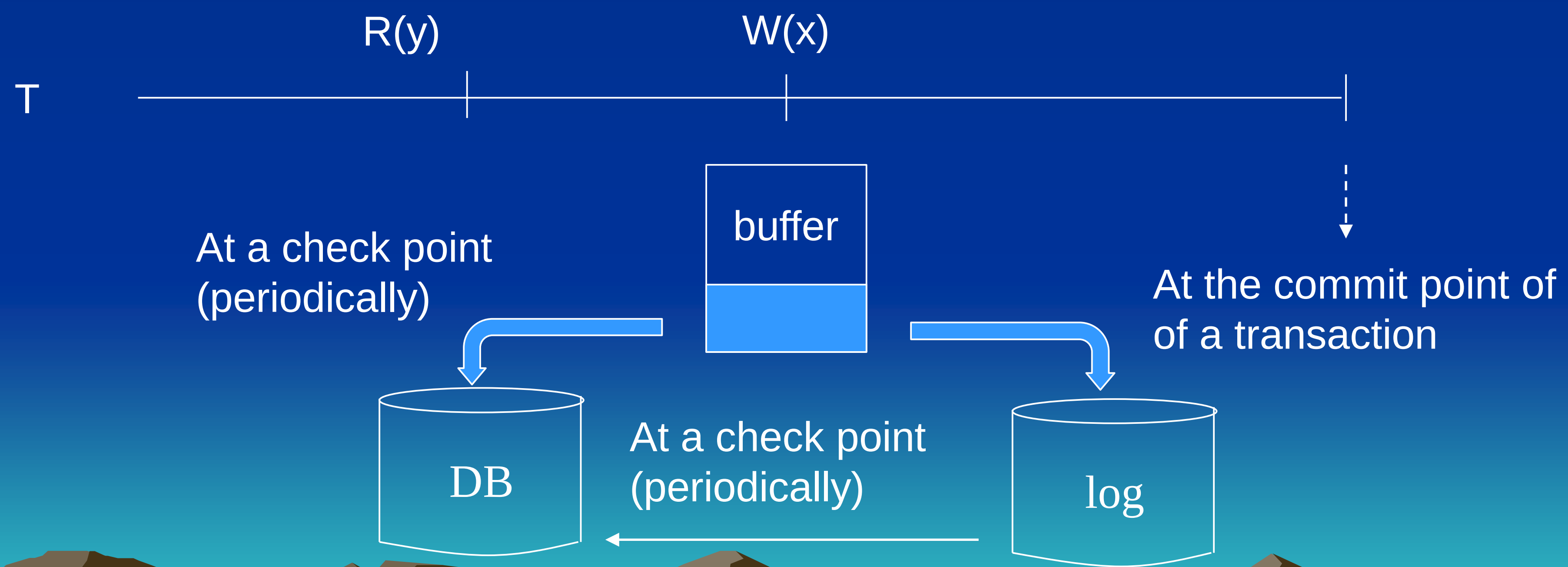
    -Allows for **redo** and **undo** operations

Commit Point

A transaction has committed when it reaches its Commit Point (when the commit command is explicitly performed).

At this point:

- The DBMS force-writes all changes/updates made by a transaction to the log
- Then the DBMS force-writes a commit record for the transaction



Checkpoint

A DBMS will execute a checkpoint in order to simplify the recovery process. The checkpoints occur periodically, arranged by a DBA (DB Administrator).

At a checkpoint any committed transactions will have their database writes (updates/changes) physically written to the database.
(The changes made by unaccomplished transactions may also be written to the database.)

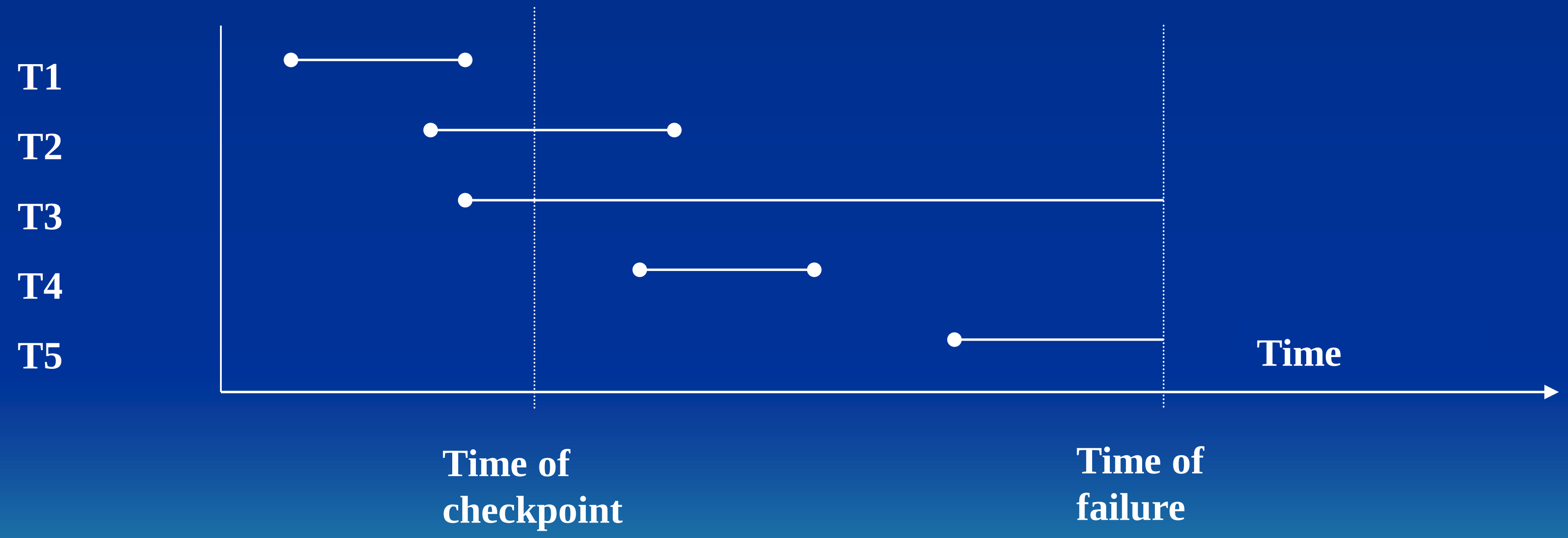
This is a four-step process

- Suspend transaction execution temporarily
- The DBMS force-writes all database changes to the database
- The DBMS writes a checkpoint record to the log and force-writes the log to disk
- Transaction execution is resumed

Transaction types at recovery time

After a system crash some transactions will need to be redone or undone.

Consider the five types below. Which need to be redone/undone after the crash?



No redo, no undo
redo
undo
redo
ignored

Transactions States

Consider the following state transition diagram

Transaction Processing

Schedule or History

- order of execution of operations of concurrent transactions
- example

  S: $R_2(X); W_2(X); R_1(X); R_1(Y); R_2(Y); W_2(Y); C_1; C_2;$

  where

  R - READ

  W - WRITE

  C - COMMIT

  A - ABORT

T1: $R_1(X); R_1(Y); C_1;$

T2: $R_2(X); W_2(X); R_2(Y); W_2(Y); C_2;$

Schedule or History

S_a: $R_1(X)$; $R_2(X)$; $W_1(X)$; $R_1(Y)$; $W_2(X)$; $W_1(Y)$; C_1; C_2;

Time	Transaction1	Transaction2	
1	READ(X)		$R_1(X)$
2	X:=X-N		
3		READ(X)	$R_2(X)$
4		X:=X+M	
5	WRITE(X)		$W_1(X)$
6	READ(Y)		$R_1(Y)$
7.		WRITE(X)	$W_2(X)$
8.	Y:=Y+N		
9.	WRITE(Y)		$W_1(Y)$
10.	Commit		C_1
11		Commit	C_2

Schedule or History

S_b: $R_1(X)$; $W_1(X)$; $R_2(X)$; $W_2(X)$; $R_1(Y)$; A_1; C_2;

Time	Transaction1	Transaction2	
1	READ(X)		$R_1(X)$
2	X:=X-N		
3	WRITE(X)		$W_1(X)$
4		READ(X)	$R_2(X)$
5		X:=X+M	
6		WRITE(X)	$W_2(X)$
7	READ(Y)		$R_1(Y)$
8	terminates!		A_1

Conflict

Two operations in a schedule conflict if they belong to two different transactions, are accessing the same data item X and one of the operations is a WRITE.

Examples:

$R_1(X)\ W_2(X)$ in: $R_1(X); W_1(X); R_2(X); W_2(X); R_1(Y); A_1; C_2;$

$W_1(X)\ W_2(X)$ in: $R_1(X); W_1(X); R_2(X); W_2(X); R_1(Y); A_1; C_2;$

$W_1(X)\ R_2(X)$ in: $R_1(X); W_1(X); R_2(X); W_2(X); R_1(Y); A_1; C_2;$

Cascading rollback:

- An uncommitted transaction has to be rolled back because it reads an item from a transaction that fails.

  Example:

  S_e: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$;$W_1(Y)$; A_1; A_2;
- Time consuming
- Avoided if there is a rule that a transaction can only read items that were written by committed transactions.

Time	Transaction1	Transaction2
1	$R_1(X)$	
2	$W_1(X)$	
3		$R_2(X)$
4	$R_1(Y)$	
5		$W_2(X)$
6	$W_1(Y)$	
7	abort	
8		abort

Complete Schedule

- A schedule S for transactions $T = \{T_1, T_2, \ldots, T_N\}$ is complete if
  - all the operations are exactly those for all transactions in the set T including Commit or Abort as the last operation of each.
  - The order of appearance of operations in S for any T_i in $\{T_1\ T_2 \ldots T_N\}$ is the same as their appearance in T_i .

S: $R_2(X)$; $W_2(X)$; $R_1(X)$; $R_1(Y)$; $R_2(Y)$; $W_2(Y)$; C_1; C_2;

⇕

T1: $R_1(X)$; $R_1(Y)$; C_1;
T2: $R_2(X)$; $W_2(X)$; $R_2(Y)$; $W_2(Y)$; C_2;

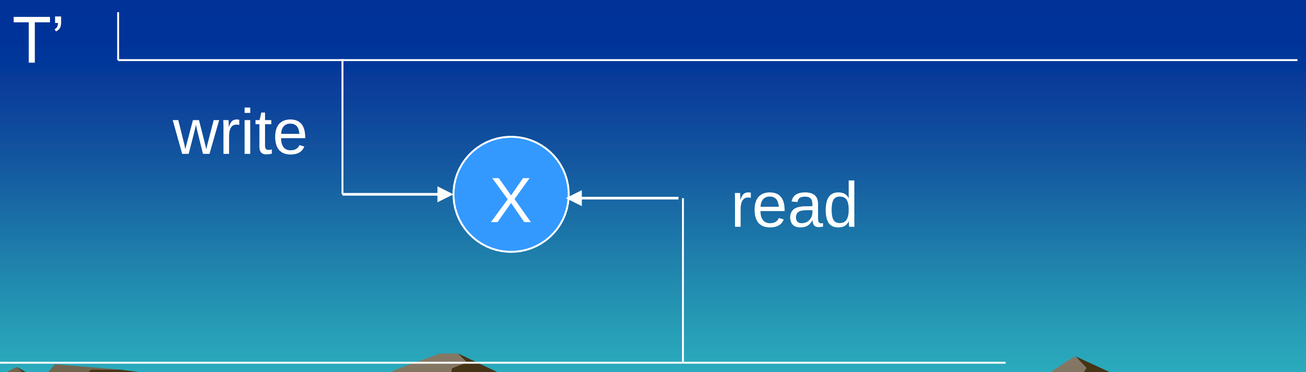
Recoverable Schedule

Recoverable: (Once a transaction is committed, it should never be necessary to roll back.)

A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

The meaning of "transaction T reads another transaction T'":

A transaction T reads from transaction T' in a schedule S if some item X is first written by T' and then read by T; and T' should not have been aborted before T reads item X, and there should be no transactions that writes X after T' writes it and before T reads it.



Recoverable Schedule

Example (Recoverable schedules):

S_a: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$; C_2; A_1;

S_b: $R_1(X)$; $R_2(X)$; $W_1(X)$; $R_1(Y)$; $W_2(X)$; C_2;$W_1(Y)$; C_1;

Recoverable Schedule

Example (Recoverable schedules):

S_a: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$; C_2; A_1;

(non-recoverable)

$R_1(X)$; $W_1(X)$; $R_1(Y)$; A_1;

$R_2(X)$; $W_2(X)$; C_2;

S_b: $R_1(X)$; $R_2(X)$; $W_1(X)$; $R_1(Y)$; $W_2(X)$; C_2;$W_1(Y)$; C_1;

(recoverable but suffers from the lost update problem)

$R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; C_1;

$R_2(X)$; $W_2(X)$; C_2;

Recoverable Schedule

Example (Recoverable schedules):

S_a: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$; C_2; A_1;

(non-recoverable)

$R_1(X)$; $W_1(X)$;		$R_1(Y)$;	A_1;
	$R_2(X)$;	$W_2(X)$; C_2;	

S_b: $R_1(X)$; $R_2(X)$; $W_1(X)$; $R_1(Y)$; $W_2(X)$; C_2;$W_1(Y)$; C_1;

(recoverable but suffers from the lost update problem)

$R_1(X)$;		$W_1(X)$; $R_1(Y)$;	$W_1(Y)$; C_1;
	$R_2(X)$;	$W_2(X)$; C_2;	

Recoverable Schedule

Example (Recoverable schedules):

S_c: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$;$W_1(Y)$; C_1; C_2;

S_d: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$;$W_1(Y)$; A_1; A_2;

Recoverable Schedule

Example (Recoverable schedules):

S_c: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$;$W_1(Y)$; C_1; C_2;

(recoverable)

$R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; C_1;

$R_2(X)$; $W_2(X)$; C_2;

S_d: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$;$W_1(Y)$; A_1; A_2;

(recoverable but cascading rollback)

$R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; A_1;

$R_2(X)$; $W_2(X)$; A_2;

A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

Is the following schedule recoverable?

S: $R_1(X)$; $W_1(X)$; $R_1(Y)$; $R_2(X)$; $W_2(X)$; C_2; $W_1(Y)$; C_1;

$R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; C_1;

$R_2(X)$; $W_2(X)$; C_2;

Cascadeless Schedule

Cascadeless (Avoid cascading rollback):

Every transaction in the schedule reads only items that were written by committed transaction.

Example:

S_1: $R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; C_1; $R_2(X)$; $W_2(X)$; C_2;

$R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; C_1;

$R_2(X)$; $W_2(X)$; C_2;

S_2: $R_1(X)$; $W_1(X)$; $R_2(Y)$; $R_1(Y)$; $W_1(Y)$; $W_2(Y)$; C_1; $R_2(X)$; $W_2(X)$; C_2;

$R_1(X)$; $W_1(X)$; $R_1(Y)$; $W_1(Y)$; C_1;

$R_2(Y)$; $W_2(Y)$ $R_2(X)$; $W_2(X)$; C_2;

Strict Schedule

- a transaction can neither read nor write an item X until the last transaction that wrote X has committed or aborted.
- In a strict schedule, the process of undoing a W(X) operation of an aborted transaction is simply to restore the before image (BFIM or old_value).
- This strategy can not be used for recoverable or cascadeless schedules.

Example:

S_1: $R_1(X)$; $W_1(X)$; $R_2(Y)$; $W_2(Y)$; C_1; $R_2(X)$; $W_2(X)$; C_2;

S_2: $R_1(X)$; $W_1(X)$; $W_2(X)$; A_1; C_2;

Strict Schedule

- a transaction can neither read nor write an item X until the last transaction that wrote X has committed or aborted.
- In a strict schedule, the process of undoing a W(X) operation of an aborted transaction is simply to restore the before image (BFIM or old_value).
- This strategy can not be used for recoverable or cascadeless schedules.

Example:

S_1: $R_1(X)$; $W_1(X)$; $R_2(Y)$; $W_2(Y)$; C_1; $R_2(X)$; $W_2(X)$; C_2;
(strict)

S_2: $R_1(X)$; $W_1(X)$; $W_2(X)$; A_1; C_2;
(non-strict)

Strict Schedule

Example:

S_f: $R_1(X)$; $R_2(X)$; $W_1(X, 5)$; $W_2(X, 8)$; C_2; A_1;

(before the transaction, X = 9)

[write_item, T_1, X, 9, 5]

Cascadeless but not strict

T_1 is aborted, X will be restored to 9. However, X has already been changed to X = 8 by T_2. Hence, it is incorrect.

Comparison of the three schedules

increasing concurrency

decreasing difficulty of recovery

Recoverable

A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

Cascadeless

Every transaction in the schedule reads only items that were written by committed transaction.

Strict

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

Serial Schedule

- A schedule is said to be serial if the transactions execute in a non-interleaved sequence. That is, all operations for any transaction T are executed consecutively.

- A serial schedule is considered correct.

- Example

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

- Serial schedules limit concurrency. Because of the tremendous speed difference between cpu operations and I/O operations, we cannot leave the cpu idle while a transaction waits for I/O.

Serializability

- A schedule is said to be serializable if it is *equivalent* to a serial schedule

- What do we mean by equivalent?

  Text mentions *result* equivalence and *conflict* equivalence.

Result equivalence

- Two schedules are said to be *result equivalent* if they produce the same database state.
- Result equivalence is not useful to us because two different schedules could accidentally produce the same database state for one set of initial values, but not for another set.

T1	T2
read_item(x);	read_item(x);
x := x + 10;	x := x * 1.1;
write_item(x);	write_item(x);

T1 and T2 are two different transactions. When x = 100, however, they produce the same result.

Conflict equivalence

- Two schedules are said to be *conflict equivalent* if
  - they have the same operations (coming from the same set of transactions)
  - the ordering of any two conflicting operations is the same in both schedules

- Recall
  Two operations *conflict* if they belong to two different transactions, are accessing the same data item X and one of the operations is a WRITE

Conflict Serializability

A schedule S is conflict serializable if it is conflict equivalent to some serial schedule S'.

R1(X), R2(Y), W2(Y), W1(X), W2(X), C1, C2

R1(X), W1(X), C1, R2(Y), W2(Y), W2(X), C2

R1(X), R2(X), W2(X), W1(X), C1, C2

R1(X), W1(X), C1, R2(X), W2(X), C2

R1(X), R2(X), W2(X), W1(X), C1, C2

R2(X), W2(X), C2, R1(X), W1(X), C1

Testing a Schedule for Conflict Serializability

- We'll construct a graph (called a *precedence graph*) where
  - nodes represent transactions
  - edges represent dependencies between transactions
    read-write
    write-read
    write-write
  - a schedule with no cycles is conflict serializable

Testing a Schedule for Conflict Serializability

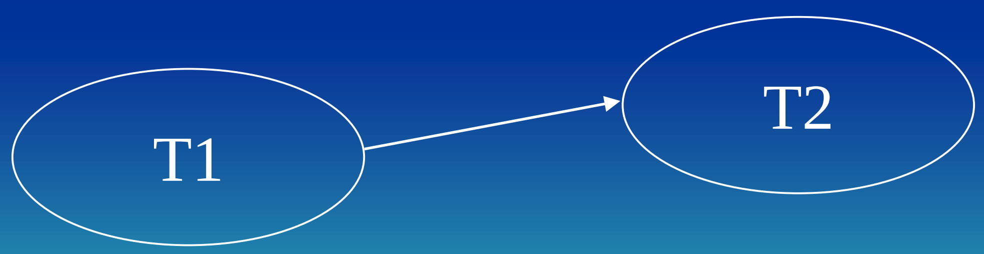
Consider a schedule S:

- For each transaction T_i in S create a node T_i in the precedence graph
- For each case in S where
  $READ_j(X)$ occurs after $WRITE_i(X)$
  create an edge $T_i \longrightarrow T_j$ in the precedence graph
- For each case in S where
  $WRITE_j(X)$ occurs after $WRITE_i(X)$
  create an edge $T_i \longrightarrow T_j$ in the precedence graph
- For each case in S where
  $WRITE_j(X)$ occurs after $READ_i(X)$
  create an edge $T_i \longrightarrow T_j$ in the precedence graph

- the schedule S is serializable if and only if the precedence graph has no cycles.

Example

Time	T1	T2
1	READ(X)	
2	X:=X-N	
3	WRITE(X)	
4		
5	READ(Y)	
6	Y:=Y+N	
7	WRITE(Y)	
8		READ(X)
9		X:=X+M
10		WRITE(X)
11		



A dependency exists between T1 and T2

But no cycles!

Example

Time	T1	T2
1	READ(X)	
2	X:=X-N	
3		READ(X)
4		X:=X+M
5	WRITE(X)	
6		
7	READ(Y)	
8		WRITE(X)
9		
10	Y:=Y+N	
11	WRITE(Y)	

The graph has a cycle!

$R_1(X);$ $W_1(X);$ $R_1(Y);$ $W_1(Y);$

$R_2(X);$ $W_2(X);$

$R_1(X);$ $W_1(X);$ $R_1(Y);$ $W_1(Y);$

$R_2(X);$ $W_2(X);$

Comments

- This test might be difficult to implement in practice
  - Since transactions are submitted continuously, when would a schedule begin and end?

- Theory of serializability forms the basis of protocols (rules) for a concurrency subsystem