



MONASH University

Information Technology

# FIT2094 Databases

Week 7 – Manipulating the Data

**algorithm** distributed systems **database**  
systems **computation** knowledge ma  
**design** e-business **model** data mining **int**  
distributed systems **database** software  
**computation** knowledge management **an**

# UPDATE

- Change the values of existing data
- For example, at the end of semester, change the mark and grade from 0,'NA' to the actual mark and grade.

```
UPDATE table  
SET column = (subquery) [, column = value, ...]  
[WHERE condition];
```

```
UPDATE enrolment  
SET mark = 80, grade = 'HD'  
WHERE sno=112233
```

```
UPDATE enrolment  
SET mark=85  
WHERE unit_code = (SELECT unit_code FROM unit WHERE unit_name='Databases')  
AND mark=80;
```

# DELETE

- Removing data from the database

```
DELETE FROM table  
[WHERE condition];
```

```
DELETE FROM enrolment  
WHERE sno='112233'  
AND unit_code= (SELECT unit_code FROM unit WHERE unit_name='Databases' )  
AND semester='1'  
AND year='2012';
```

# Outline

1. Introduction
2. Transaction Properties
3. Serializability
4. Locking
5. Database Recovery

# Transaction

- Consider the following situation.

*Sam is transferring \$100 from his bank account to his friend Jim's.*

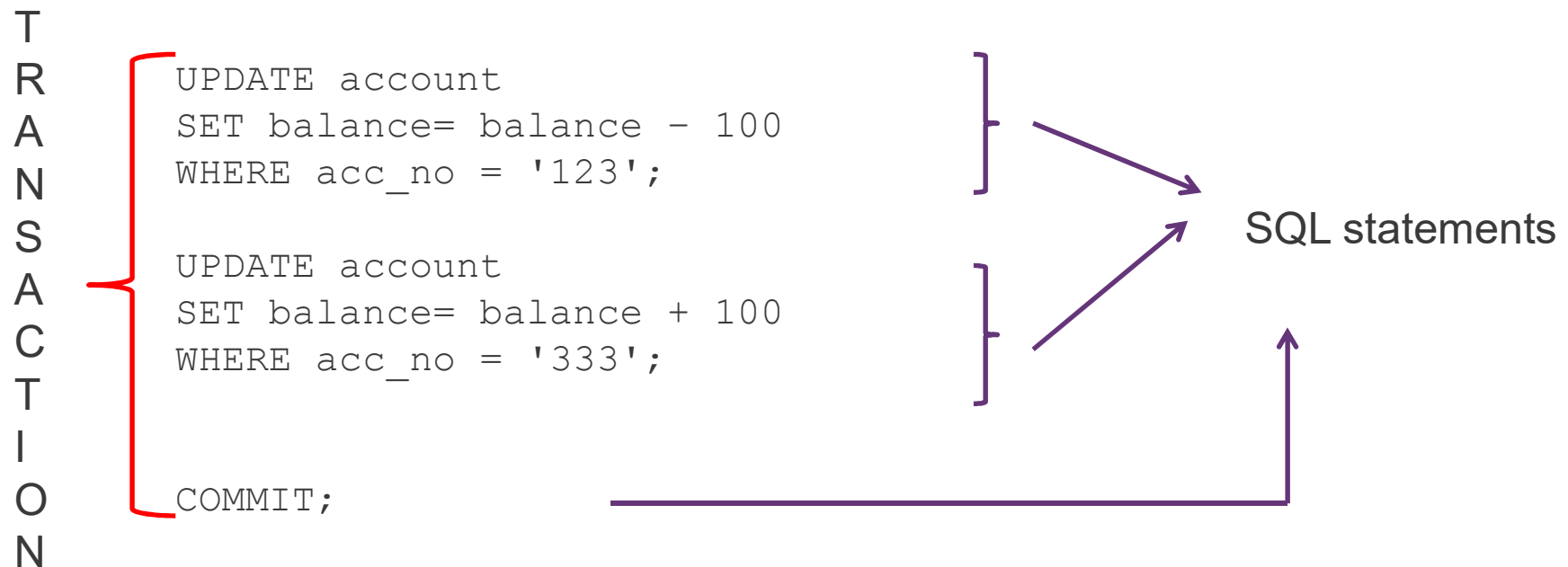
- Sam's account should be reduced by 100.
- Jim's account should be increased by 100.

***Sam's account should be reduced by 100.***

**Q4. Which of the following SQL statements is correct for the above operation? Assume Sam's account number is '123'.**

- A. UPDATE account  
SET balance = balance – 100;
- B. UPDATE account  
SET balance= balance – 100  
WHERE acc\_no = '123';
- C. UPDATE account  
SET acc\_no = balance + 100;
- D. UPDATE account  
SET balance = balance + 100  
WHERE acc\_no = '123';

Assume that Jim's account number is '333'. The transfer of money from Sam's to Jim's account will be written as the following SQL transaction:



All statements need to be run as a single logical unit operation.

# Outline

1. Introduction
2. Transaction Properties
3. Serializability
4. Locking
5. Database Recovery and Back



# Transaction Properties

- A transaction is the execution of a program that accesses (read) or changes (write) the contents of a database system.
- A transaction must have the following properties:
  - **Atomicity**
    - All database operations (SQL requests) of a transaction must be entirely completed or entirely aborted
  - **Consistency**
    - It must take the database from one consistent state to another
  - **Isolation**
    - It must not interfere with other concurrent transactions
    - Data used during execution of a transaction cannot be used by a second transaction until the first one is completed
  - **Durability**
    - Once completed the changes the transaction made to the data are durable, even in the event of system failure

**Q5. According to the *atomicity* property, the transaction below is complete when statement number \_\_\_\_\_ is completed.**

1 UPDATE account  
SET balance= balance - 100  
WHERE acc\_no = '123';

2 UPDATE account  
SET balance= balance + 100  
WHERE acc\_no = '333';

3 COMMIT;

- A. 1
- B. 2
- C. 3
- D. None of the above.

**Q6. Which transaction property is violated when a transaction T2 (Jim checking the account balance) is allowed to read the balance of Jim's account while the transaction T1 (the money transfer from Sam's to Jim's) has not been completed?**

- A. Atomicity.
- B. Isolation.
- C. Consistency.
- D. Durability.

# Consistency - Example

- Assume that the server lost its power during the execution of the money transfer transaction, only the first statement is completed (taking the balance from Sam's).
- Consistency properties ensure that Sam's account will be reset to the original balance because the money has not be transferred to Jim's account.
- The last consistent state is *when the money transfer transaction has not been started*.

# Durability - Example


- Assume the server lost power after the commit statement has been reached.
- The durability property ensures that the balance on both Sam's and Jim's accounts reflect the completed money transfer transaction.

# Transaction Management

- **Transaction management:** The management of multiple transactions submitted concurrently by various users to the same database system.
- Follows the ACID properties.
- Transaction boundaries
  - Start
    - first SQL statement is executed (eg. Oracle)
    - Some systems have a BEGIN WORK type command
  - End
    - COMMIT or ROLLBACK
- Concurrency Management

# Outline

1. Introduction
2. Transaction Properties
3. Serializability
4. Locking
5. Database Recovery



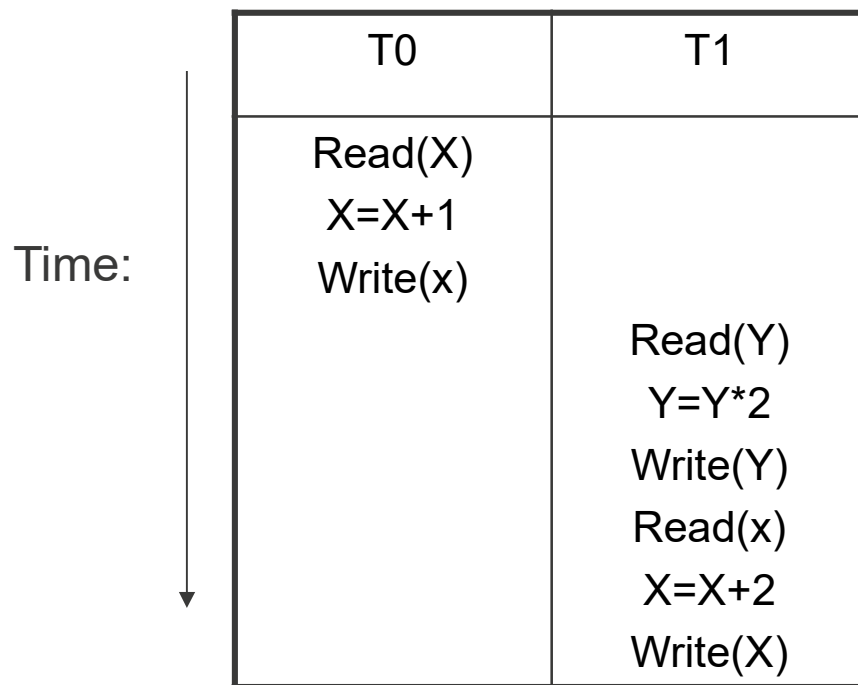
A Transaction is a logical unit of work including reads from/writes to a database, which must be **executed as a whole** or **not at all**, in order for the consistency of the database to be maintained.

Two main database operations that can be included in a transaction:

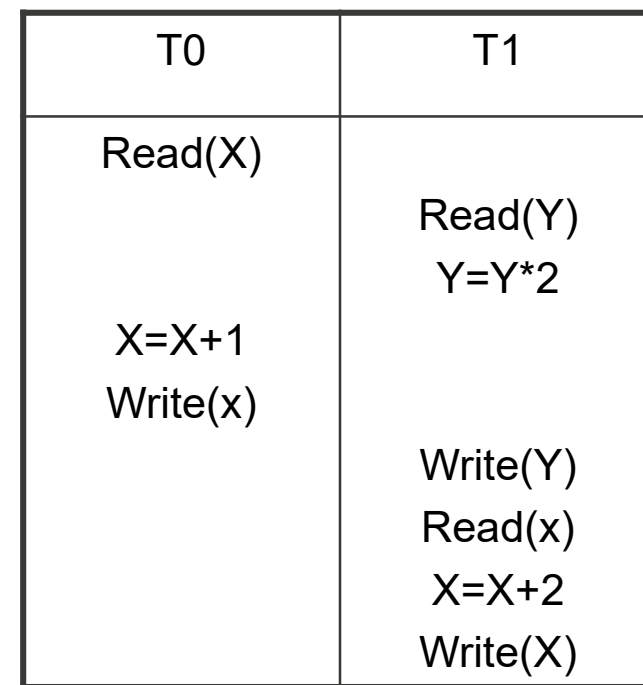
- o **Read(x)** returns the value stored in data item x.
- o **Write(x)** changes the stored value of x.



## *Serial* and *Interleaved* transactions.



***Serial***



***Interleaved***

Why transaction management for concurrent transactions is needed?

## The Lost Update Problem


T0	T1
Read(X) $X = X + 10$ Write(X)	Read(X) $X = X - 20$ Write(X)

***Serial***

T0	T1
Read(X)  $X = X + 10$ Write(X)	Read(X)   $X = X - 20$ Write(X)

***Interleaved***

*value written by T0 is lost*



## Q7. What transaction property is violated in the Lost-Updates problem?

- A. Atomicity.
- B. Consistency.
- C. Isolation.
- D. Durability.

## The Temporary Update (Dirty Read) Problem

T0	T1
Read(X) $X = X + 10$ Write(X) Read(Y) Abort	       Read(X) $X = X - 20$ Write(X)

***Serial***

T0	T1
Read(X) $X = X + 10$ Write(X)    Read(Y) Abort	    Read(X) $X = X - 20$ Write(X)

***Interleaved***

*T1 reads dirty value of X*

## The Incorrect Summary Problem


T0	T1
Read(X) X=X-10 Write(X) Read(Y) Y=Y+10 Write(Y)	          Sum = 0 Read(X) Sum = Sum + X Read(Y) Sum = Sum + Y

***Serial***

T0	T1
Read(X) X=X-10 Write(X)      Read(Y) Y=Y+10 Write(Y)	          Sum = 0 Read(X) Sum = Sum + X Read(Y) Sum = Sum + Y

***Interleaved***

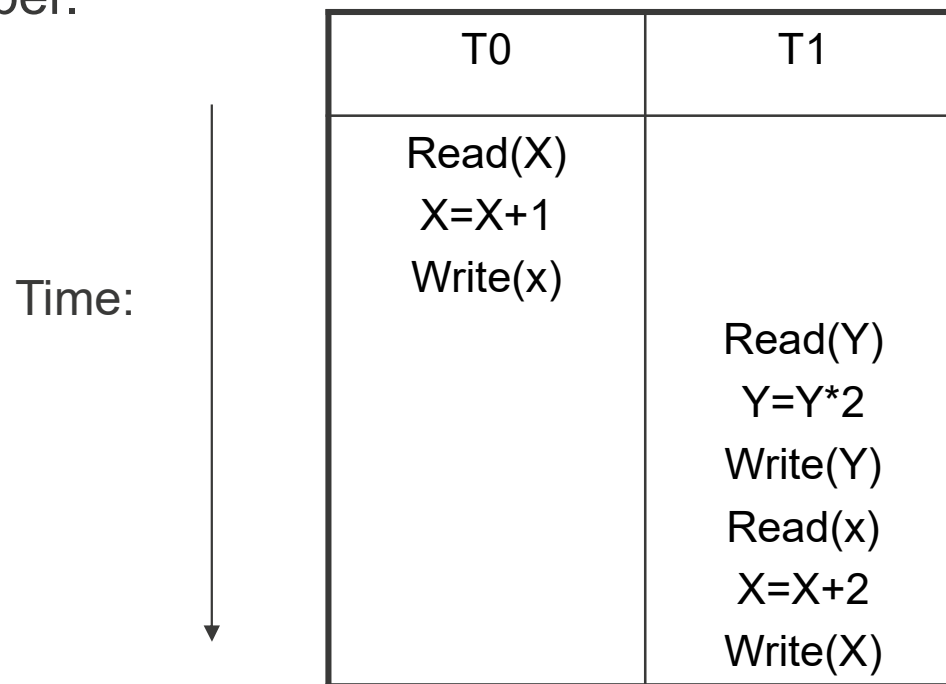
*T1 reads X **after** subtraction but Y **before** addition*



**Q8. What SQL statement would be considered to be a READ operation in concurrency management?**

- A. SELECT statement
- B. UPDATE statement
- C. DELETE statement
- D. Commit statement

- o In Transaction Management, only the following operations are considered: **Read**, **Write**, **Commit**, and **Abort**.
- o Therefore Scenario 1 (**Serial Schedule**) can be written as:  
**S1: r0(X); w0(X); c0; r1(Y); w1(Y); r1(X); w1(X); c1;**
- o Note that **r** means read, **w** means write, and **c** means commit. Other operations are omitted. 0 and 1 indicate the transaction number.



If transactions cannot be interleaved, there are two possible correct schedules: one that has all operations in T0 before all operations in T1, and a second schedule that has all operations in T1 before all operations in T0 (these are known as **serial schedules**).

Serial Schedule 1: r0(X); w0(X); c0; r1(Y); w1(Y); r1(X); w1(X); c1;

Serial Schedule 2: S1: r1(Y); w1(Y); r1(X); w1(X); c1; r0(X); w0(X); c0;

If transactions can be interleaved, there may be many possible interleavings (**nonserial schedules**)

Interleaved Schedule 1:

r0(X); r1(Y); w0(X); w1(Y); r1(X); c0; w1(X); c1;

Interleaved Schedule 2:

S3: r0(X); r1(Y); w1(Y); r1(X); w1(X); c1; w0(X); c0;



“A given interleaved execution of some set of transactions is said to be **serializable** if and only if it produces the same result as some serial execution of those same transactions”.

T0	T1
Read(X) X=X+1 Write(X)	Read(Y) Y=Y*2 Write(Y) Read(X) X=X+2 Write(X)


**Serial**

T0	T1
Read(X)   X=X+1 Write(X)	Read(Y) Y=Y*2 Write(Y)  Read(X) X=X+2 Write(X)

**Interleaved**  
*This is serializable*

T0	T1
Read(X)   X=X+1 Write(X)	Read(Y) Y=Y*2 Write(Y) Read(X) X=X+2  Write(X)

**Interleaved**  
*This is NOT serializable*



“A given interleaved execution of some set of transactions is said to be **serializable** if and only if it produces the same result as some serial execution of those same transactions”.

For interleaved schedules, we need to determine whether the interleaved schedules are correct, i.e., are **serializable**.

*“Serializable schedules are equivalent to a serial schedule of the same transactions”.*

**Serializability Theory**: an important theory of concurrency control which attempts to determine which schedules are “correct” and which are not and to develop techniques that allow only correct schedules.

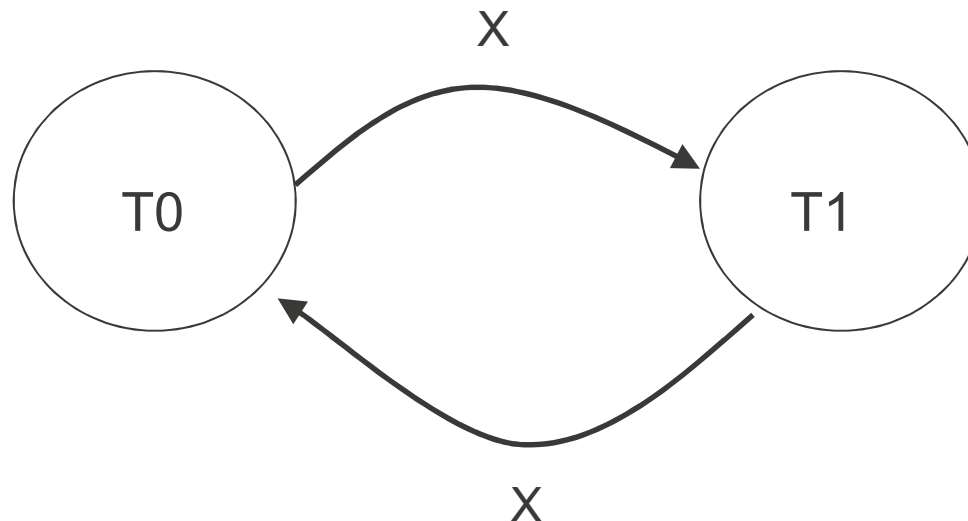
## Precedence Graph

One approach to test for the serializability of transactions is to draw a directed graph (**precedence graph**) using the following approach:

1. For each transaction  $T_i$  in the schedule, draw a node labelled  $T_i$ .
2. For each conflicted operation where  $T_j$  occurs after  $T_i$ , draw an edge  $T_i \rightarrow T_j$ .
3. The schedule is serializable, iff there are no cycles (acyclic).

<b>T0</b>	<b>T1</b>	<b>Result</b>
Read(X)	Read(X)	No conflict
Read(X)	Write(X)	Conflict
Write(X)	Read(X)	Conflict
Write(X)	Write(X)	Conflict
Read(X)	Write(Y)	No conflict (different items)

“Is **S2**: **r0(X); r1(Y); w1(Y); r1(X); w1(X); c1; w0(X); c0**; serializable?”



The edge from T0 to T1 is present since  $w1(X)$  occurs after  $r0(X)$ . The edge from T1 to T0 is present since  $w0(X)$  occurs after  $r1(X)$ .

Is this serializable?

T0	T1
Read(X)	Read(Y)
	Write(Y)
Write(X)	Read(X)
	Write(X)

Is this serializable?

T0	T1
Read(X)	Read(Y) Write(Y) Read(X)
Write(X)	Write(X)

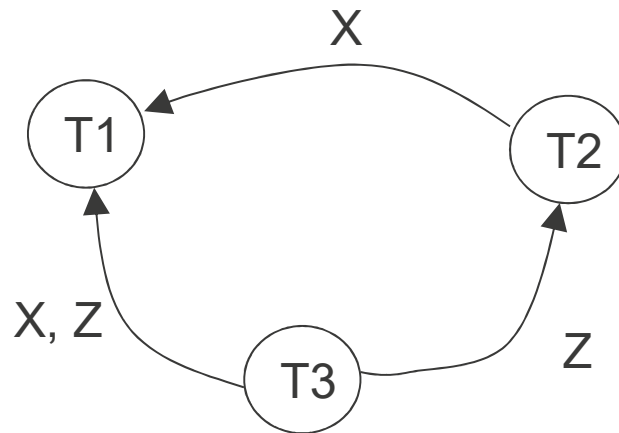
## Equivalent Serializable Schedules

If the graph contains no cycles, we can create an equivalent serial schedule by:

- o Identifying each edge in the graph.
- o Ordering the schedule so that for an edge  $T_i \rightarrow T_j$ ,  $T_i$  occurs before  $T_j$ .
- o If there are no edges, then the operations can be ordered in (almost) any order.

## An Example (Interleaved Schedule):

S1:  $r_1(X)$ ;  $r_2(X)$ ;  $r_3(X)$ ;  $r_2(Y)$ ;  $w_2(Y)$ ;  $r_3(Z)$ ;  $w_3(Z)$ ;  $r_1(Z)$ ;  $w_1(X)$ ;  
 $r_2(Z)$ ;  $c_2$ ;  $c_1$ ;  $c_3$ ;



## Equivalent Serialized Schedule:

S1:  $r_3(X)$ ;  $r_3(Z)$ ;  $w_3(Z)$ ;  $c_3$ ;  $r_2(X)$ ;  $r_2(Y)$ ;  $w_2(Y)$ ,  $r_2(Z)$ ;  $c_2$ ;  $r_1(X)$ ;  
 $r_1(Z)$ ;  $w_1(X)$ ;  $c_1$ ;



# Outline

1. Introduction
2. Transaction Properties
3. Serializability
4. Locking
5. Database Recovery

## ■ Locking

A mechanism to overcome the problems caused by non-serialized transactions (i.e. lost update, temporary update, and incorrect summary problems).

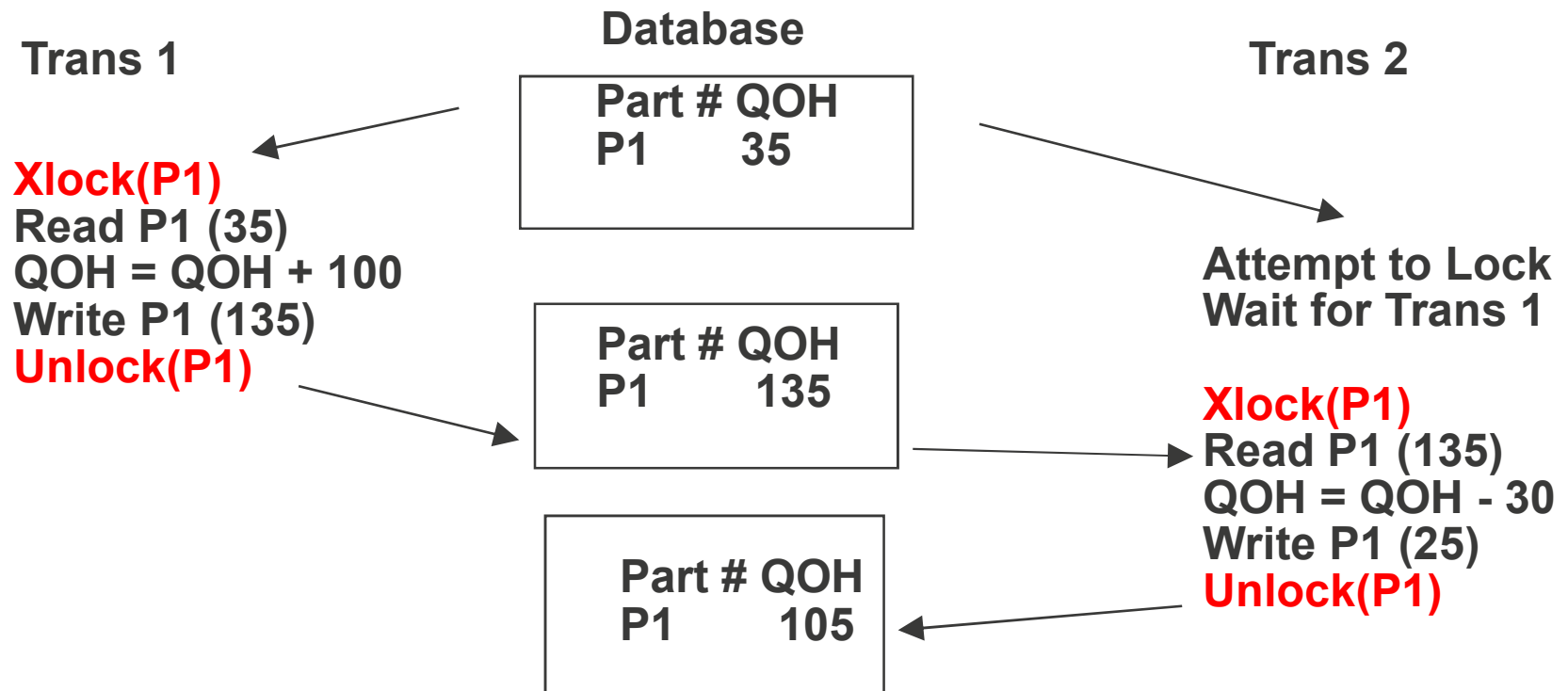
Locking is used to synchronize transaction execution, and in particular enforce transactions to be serializable.

## ■ Locking Scheme:

- read\_lock(x) - shared lock
- write\_lock(x) - exclusive lock

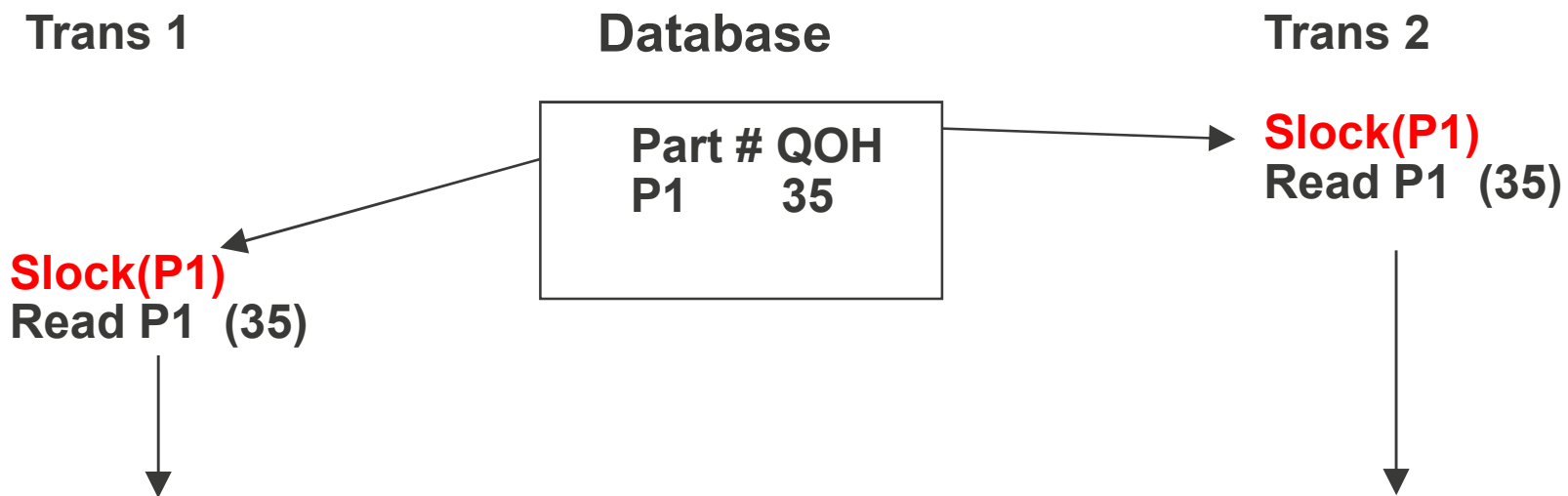
# Exclusive Locks – Example 1

- Write-locked items
  - require an Exclusive Lock
  - a single transaction exclusively holds the lock on the item



## Shared Locks – Example 2

- Read-locked items
  - require a Shared Lock
  - allows other transactions to read the item



- **Shared locks** improve the amount of concurrency in a system  
If **Trans 1** and **Trans 2** only wished to read **P1** with no subsequent update they could both apply an **Slock** on **P1** and continue


# Lock Granularity

- Granularity of locking refers to the size of the units that are, or can be, locked. Locking can be done at
  - database level
  - table level
  - page level
  - **record level**
    - Allows concurrent transactions to access different rows of the same table, even if the rows are located on the same page.
  - attribute level
    - Allows concurrent transactions to access the same row, as long as they require the use of different attributes within that row.

## Two-Phase Locking

- Guaranteeing serializability by using Two Phase Locking.
- A transaction is said to follow the **Two Phase Locking** protocol if ALL **lock** operations (read and write locks) precede the first **unlock** operation in the transaction. Such a transaction can be divided into two phases:
  - **Growing Phase**, during which new locks on items can be acquired but none can be released;
  - **Shrinking Phase**, during which existing locks can be released but no new locks can be acquired.

- When a DBMS first receives a lock request for  $T_i$ , it checks to see if the lock conflicts with other lock by other transaction/s.
  - If so, it delays the request, forcing  $T_i$  to **wait** until it can set the lock it needs.
  - If not, the DBMS obtains the requested lock and executes the operation.
- Once a DBMS has set the lock for  $T_i$ , it may **not release that lock** at least until the corresponding operation has executed.
- Once a DBMS has released a lock for a transaction, it **may not subsequently obtain any more locks** for that transaction on any data item.

- 
- There are three different types of two phase locking:
    - Basic 2PL,
    - Strict 2PL, and
    - Conservative 2PL.



## Basic 2PL

In Basic 2PL there is a *growing phase* when locks are obtained and a *shrinking phase* when locks are released.

$T_1$	X and Y values for initial X=20, Y=30
read_lock(Y);	
read_item(Y);	Y=30
write-lock(X);	
unlock(Y);	
read_item(X);	X=20
X:=X+Y;	
write_item(X)	X=50
unlock(X);	



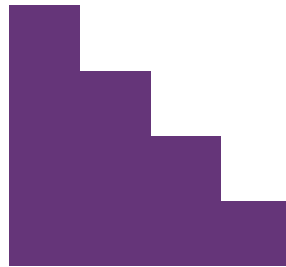
## Strict 2PL

- The most popular 2PL.
- All the locks are released together when the transaction terminates.
- Since the locks will be released after **COMMIT** or **ABORT**, then it is certain that all lock requests have been issued and no further actions against data items in potential conflict will occur.



## Conservative 2PL

- Acquire all necessary locks in advance before starting the transaction.
- Release the locks one by one as the transaction goes along.



## Locking theory to avoid Lost Update Problem

T0	T1
Read(X)	Read(X)
X=X+10	
Write(X)	
	X=X-20
	Write(X)

T1 has a read lock on X,  
so T0 cannot write  
unless lock released by  
T0

***Interleaved***

## Locking theory to avoid Incorrect Summary Problem

T0	T1
Read(X) X=X-10 Write(X)	Sum = 0 Read(X) Sum = Sum + X Read(Y) Sum = Sum + Y
Read(Y) Y=Y+10 Write(Y)	

Since T0 cannot release a lock until it has acquired all locks, T1 needs to wait to read X until T0 also locks Y. In this case, T0 reads updated value of X and cannot read Y unless the lock on Y is released by T0 after updating Y.

## Deadlock problem?

T0	T1
Read(X) Write(X)  Read(Y)  Write(X)	  Read(Y) Write(Y)  Read(X) Write(Y)

T0 is waiting for lock on Y

T1 is waiting for lock on X

No lock is released, both transactions keep waiting...

## Deadlock Detection

- **Deadlock** occurs when each of the two transactions is waiting for the other to release the lock on an item.
- Deadlock can be detected by using the Precedence Graph.
- Precedence Graph is also used to test for conflict serializability of a schedule.
- A schedule has no deadlock, if and only if the precedence graph has no cycles.

## Deadlock Resolution

- If we discover that the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transaction to abort is called as **victim selection**.
- The algorithm for **victim selection** should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and should try instead to ***select transactions that have not made any changes or that involved in more than one deadlock cycle in the wait-for graph.***



T1

---

Read\_lock(x)  
(acquire read lock)  
Read(x)  
 $x = x - n$

Write\_lock(x)  
(request x lock)  
wait  
wait  
wait  
wait  
(acquired x lock)  
*continue the process*

T2

---

Read\_lock(x) (acquire read lock)  
 $x = x + m$

Write\_lock(x)  
(request x lock)  
wait  
wait  
*Killed by the system and Rollback*

## Exercises

- Examine the following transactions. Draw a precedence graph and identify which transaction(s) produce a deadlock.

### Exercise 1:

T1 read(x)

T1 write(x)

T2 read(y)

T2 write(y)

T1 write(z)

Commit T1

Commit T2

## Exercises

- Examine the following transactions. Draw a precedence graph and identify which transaction(s) produce a deadlock.

### Exercise 1:

T1 read(x)

T1 write(x)

T2 read(y)

T2 write(y)

T1 write(z)

Commit T1

Commit T2

Watch MULO Recording for the solution...

## Exercises

- Examine the following transactions. Draw a precedence graph and identify which transaction(s) produce a deadlock.

### Exercise 1:

T1 read(x)

T1 write(x)

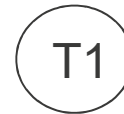
T2 read(y)

T2 write(y)

T1 write(z)

Commit T1

Commit T2



## Exercise 2:

T1 read(x)  
T1 write(x)  
T2 write(x)  
T2 read(y)  
T1 read(y)  
T1 write(x)  
Commit T1  
Commit T2

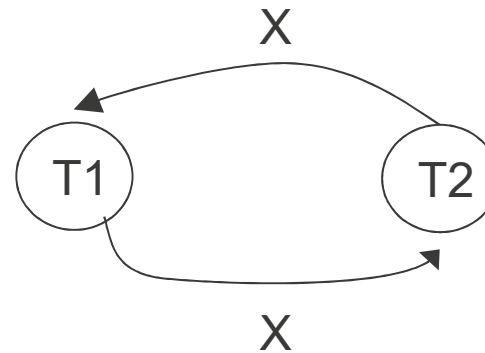
## Exercise 2:

T1 read(x)  
T1 write(x)  
T2 write(x)  
T2 read(y)  
T1 read(y)  
T1 write(x)  
Commit T1  
Commit T2

Watch MULO Recording for the solution...

## Exercise 2:

T1 read(x)  
T1 write(x)  
T2 write(x)  
T2 read(y)  
T1 read(y)  
T1 write(x)  
Commit T1  
Commit T2



### Exercise 3:

T1 read(x)  
T2 read(x)  
T3 read(x)  
T2 read(y)  
T2 write(y)  
T3 read(z)  
T3 write(z)  
T1 read(z)  
T1 write(x)  
T2 read(z)  
Commit T2  
Commit T1  
Commit T3



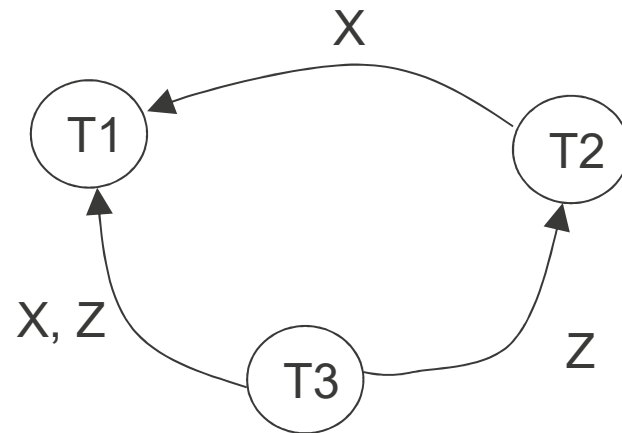
### Exercise 3:


T1 read(x)  
T2 read(x)  
T3 read(x)  
T2 read(y)  
T2 write(y)  
T3 read(z)  
T3 write(z)  
T1 read(z)  
T1 write(x)  
T2 read(z)  
Commit T2  
Commit T1  
Commit T3

Watch MULO Recording for the solution...


### Exercise 3:

T1 read(x)  
T2 read(x)  
T3 read(x)  
T2 read(y)  
T2 write(y)  
T3 read(z)  
T3 write(z)  
T1 read(z)  
T1 write(x)  
T2 read(z)  
Commit T2  
Commit T1  
Commit T3





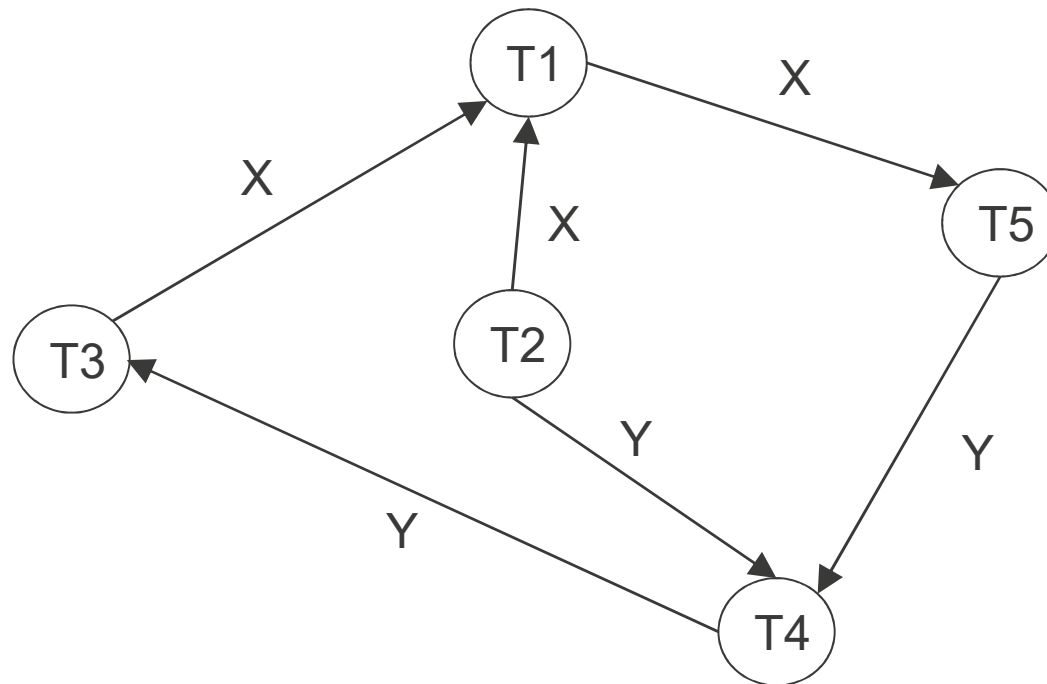
**Exercise 4a:**  $r1(X)$ ;  $r2(Y)$ ;  $r2(X)$ ;  $r3(X)$ ;  $r5(Y)$ ;  $r4(Y)$ ;  $w1(X)$ ;  $w4(Y)$ ;  $r5(X)$ ;  
 $r3(Y)$ ;  $c1$ ;  $c2$ ;  $c3$ ;  $c4$ ;  $c5$ .



**Exercise 4a:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); w1(X); w4(Y); r5(X);$   
 $r3(Y); c1; c2; c3; c4; c5.$

Watch MULO Recording for the solution...

**Exercise 4a:**  $r1(X)$ ;  $r2(Y)$ ;  $r2(X)$ ;  $r3(X)$ ;  $r5(Y)$ ;  $r4(Y)$ ;  $w1(X)$ ;  $w4(Y)$ ;  $r5(X)$ ;  $r3(Y)$ ;  $c1$ ;  $c2$ ;  $c3$ ;  $c4$ ;  $c5$ .



**Exercise 4a:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); w1(X); w4(Y); r5(X);$   
 $r3(Y); c1; c2; c3; c4; c5.$

**Exercise 4b:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); \text{c2}; w1(X); w4(Y);$   
 $r5(X); r3(Y); c1; c3; c4; c5.$

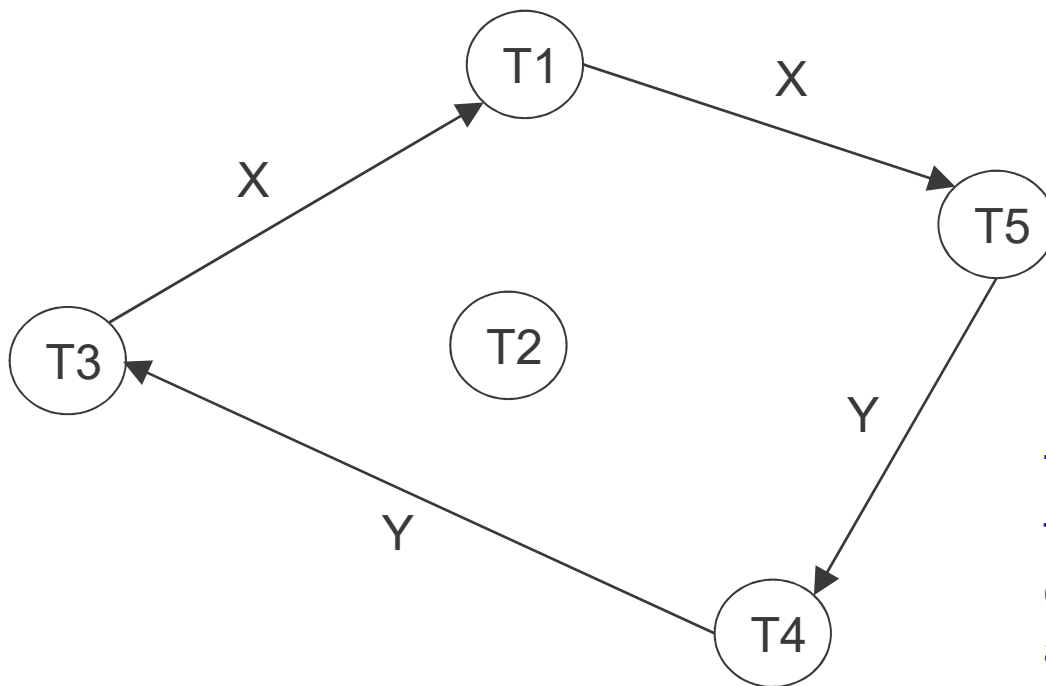
**Exercise 4a:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); w1(X); w4(Y); r5(X);  
r3(Y); c1; c2; c3; c4; c5.

**Exercise 4b:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); **c2**; w1(X); w4(Y);  
r5(X); r3(Y); c1; c3; c4; c5.

Watch MULO Recording for the solution...

**Exercise 4a:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); w1(X); w4(Y); r5(X);$   
 $r3(Y); c1; c2; c3; c4; c5.$

**Exercise 4b:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); \text{c2}; w1(X); w4(Y);$   
 $r5(X); r3(Y); c1; c3; c4; c5.$



T2 that has made T1 and T4 wait, has finished (committed), hence, T1 and T4 can continue (and T2 is not waiting for others anyway, so T2 can finish successfully) However, the cycle  $T1 \rightarrow T5 \rightarrow T4 \rightarrow T3 \rightarrow T1$  still exists.



**Exercise 4a:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); w1(X); w4(Y); r5(X);  
r3(Y); c1; c2; c3; c4; c5.

**Exercise 4b:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); **c2**; w1(X); w4(Y);  
r5(X); r3(Y); c1; c3; c4; c5.

**Exercise 4c:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); **c2**; w1(X); **c1**; w4(Y);  
r5(X); r3(Y); c3; c4; c5.

**Exercise 4a:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); w1(X); w4(Y); r5(X);  
r3(Y); c1; c2; c3; c4; c5.

**Exercise 4b:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); **c2**; w1(X); w4(Y);  
r5(X); r3(Y); c1; c3; c4; c5.

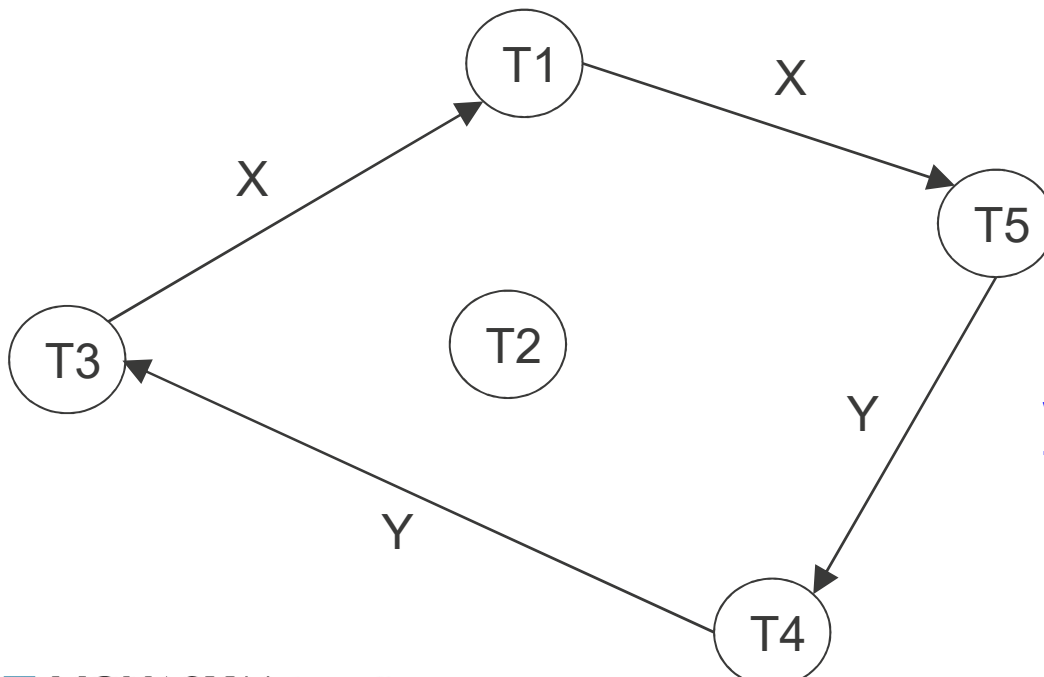
**Exercise 4c:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); **c2**; w1(X); **c1**; w4(Y);  
r5(X); r3(Y); c3; c4; c5.

Watch MULO Recording for the solution...

**Exercise 4a:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); w1(X); w4(Y); r5(X);$   
 $r3(Y); c1; c2; c3; c4; c5.$

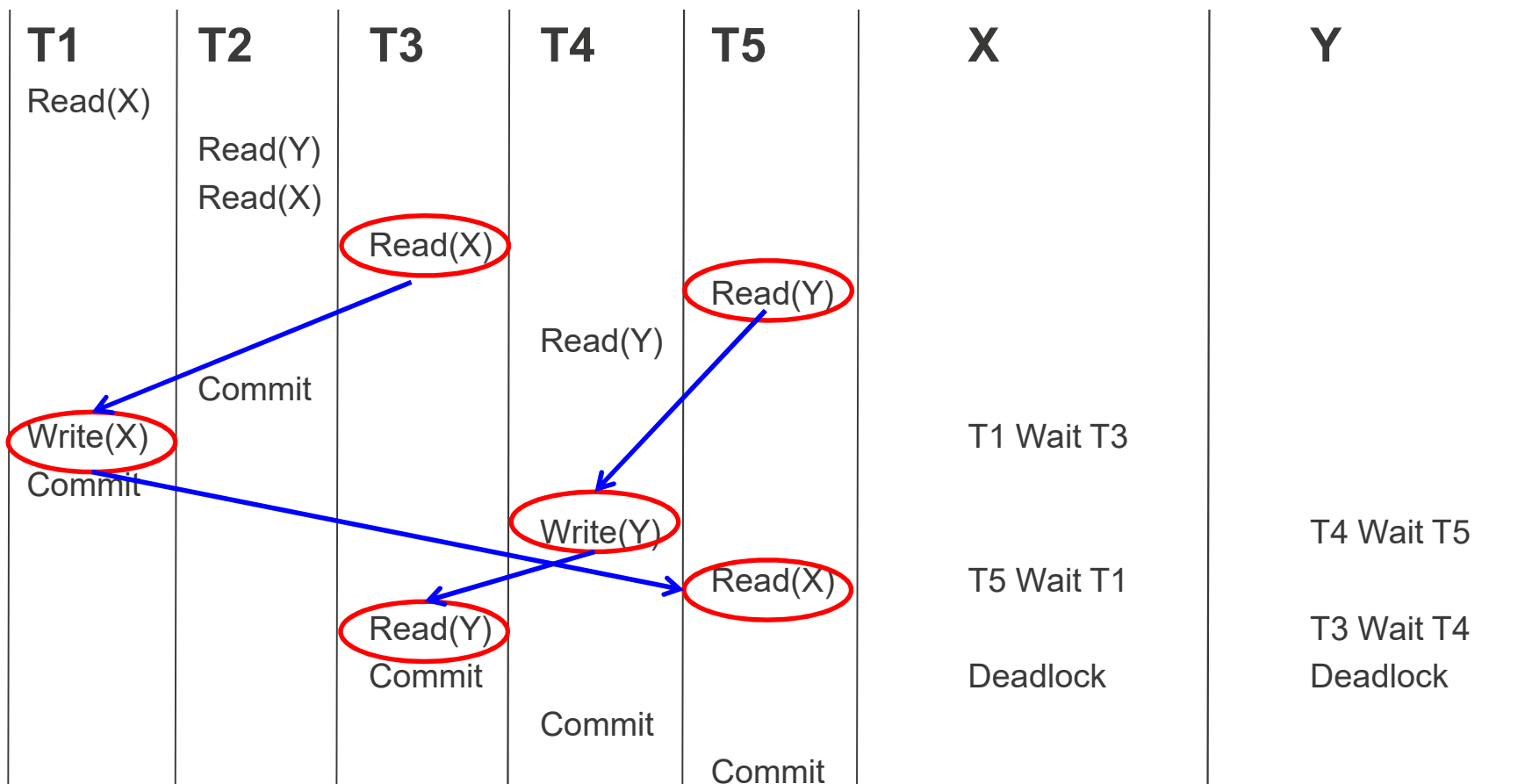
**Exercise 4b:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); \mathbf{c2}; w1(X); w4(Y);$   
 $r5(X); r3(Y); c1; c3; c4; c5.$

**Exercise 4c:**  $r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); \mathbf{c2}; w1(X); \mathbf{c1}; w4(Y);$   
 $r5(X); r3(Y); c3; c4; c5.$



C1 cannot be done, because  $w1(X)$  is still waiting for T3 to complete (that is  $r3(X)$ ). The precedence graph shows a cycle.

**Exercise 4c:** r1(X); r2(Y); r2(X); r3(X); r5(Y); r4(Y); **c2**; w1(X); **c1**; w4(Y);  
r5(X); r3(Y); c3; c4; c5.



# Outline

1. Introduction
2. Transaction Properties
3. Serializability
4. Locking
5. Database Recovery

# Database Restart and Recovery

- Restart
  - Soft crashes
    - loss of volatile storage, but no damage to disks. These necessitate restart facilities.
- Recovery
  - Hard crashes
    - hard crashes - anything that makes the disk permanently unreadable. These necessitate recovery facilities.
- Requires transaction log.

# Transaction Log

- The **log**, or journal, tracks all transactions that update the database. It stores
  - For each transaction component (SQL statement)
    - Record for beginning of transaction
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - “Before” and “after” values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
    - The ending (COMMIT) of the transaction
- The **write-ahead-log protocol** is followed, i.e., log is written **before** any data is updated in the database.
- The log should be written to a **multiple** separate physical devices.

# Checkpointing

- Although there are a number of techniques for checkpointing, the following explains the general principle. A checkpoint is taken regularly, say every 15 minutes, or every 20 transactions.
- The procedure is as follows:
  - Accepting new transactions is temporarily halted, and current transactions are suspended.
  - Results of committed transactions are made permanent (force-written to the disk).
  - A checkpoint record is written in the log.
  - Execution of transactions is resumed.

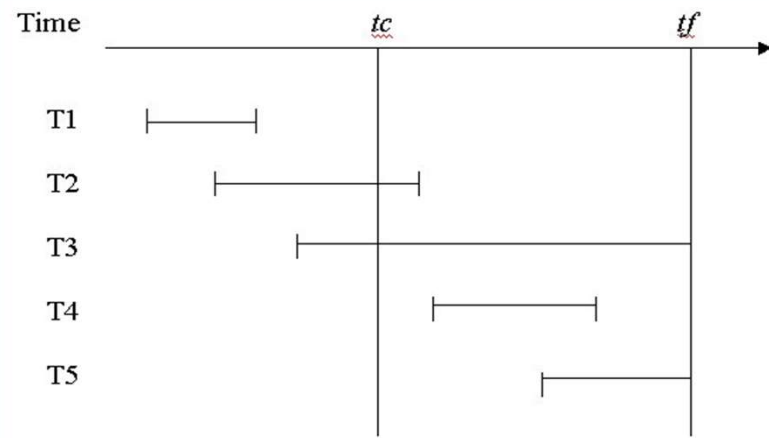


# Write Through Policy

- The database is immediately updated by transaction operations during the transaction's execution, before the transaction reaches its commit point
- If a transaction aborts before it reaches its commit point a ROLLBACK or UNDO operation is required to restore the database to a consistent state
- The UNDO (ROLLBACK) operation uses the log before values

# Restart Procedure for Write Through

- Once the cause of the crash has been rectified, and the data base is being restarted:
  - The last checkpoint before the crash in the log file is identified. It is then read forward, and two lists are constructed:
  - a REDO list containing the transaction-ids of transactions that were committed.
  - and an UNDO list containing the transaction-ids of transactions that never committed
- The data base is then rolled forward, using REDO logic and the after-images and rolled back, using UNDO logic and the before-images.



$tc$  = time of checkpoint  
 $tf$  = time of failure

**Q7. What transaction will need to be REDONE (in the REDO list)?**

- A. T1 and T2.
- B. T2 and T4.
- C. T2 and T5.
- D. T1, T2 and T3.
- E. None of the above.

## **An alternative - Deferred Write**

- The database is updated only after the transaction reaches its commit point
- Required roll forward (committed transactions redone) but does not require rollback

# Recovery

- A hard crash involves physical damage to the disk, rendering it unreadable. This may occur in a number of ways:
  - Head-crash. The read/write head, which normally “flies” a few microns off the disk surface, for some reason actually contacts the disk surface, and damages it.
  - Accidental impact damage, vandalism or fire, all of which can cause the disk drive and disk to be damaged.
- After a hard crash, the disk unit, and disk must be replaced, reformatted, and then re-loaded with the data base.

# Backup

- A backup is a copy of the data base stored on a different device to the data base, and therefore less likely to be subjected to the same catastrophe that damages the data base. (NOTE: A backup is not the same as a checkpoint.)
- Backups are taken say, at the end of each day's processing.
- Ideally, two copies of each backup are held, an on-site copy, and an off-site copy to cater for severe catastrophes, such as building destruction.
- Transaction log – backs up only the transaction log operations that are not reflected in a previous backup of the database.

# Recovery

- Re-build the data base from the most recent backup.  
This will restore the data base to the state it was in say, at close-of-business yesterday.
- **REDO** all committed transactions up to the time of the failure - no requirement for **UNDO**