Bases de données

Prof. Giovanna Di Marzo Serugendo
Assistant: Mohammad Parhizkar
mohammad.parhizkar@unige.ch

Semestre: printemps 2019

session 8- 06/05/2019

Agenda

1. ***Transactions***
    - *Transactions: Definitions, Facts and Examples*
    - *Transactions: Implementation in Oracle with an Example*
    - *Transaction Properties: ACID*
    - *Concurrency Control, Schedules: Synchronization of Transactions*
        - *Two Types of Transaction Schedules: Serial Schedule, Concurrent Schedule*
        - *Problem of Transaction Schedules*
2. ***Serializability***
    - *Conflict Equivalent, Conflict Serializability*
    - *Conflict Graph = Precedence Graph = Serializability Graph*
    - *Recoverability of a Schedule*
3. ***Locks in DBMS***
    - *Exclusive Locks vs Shared Locks*
    - *Locks Compatibility*
    - *Locks Protocols:*
        - *2-phase Locking (2PL) Protocol*
        - *Strict 2-phase Locking (strict-2PL) Protocol*
4. ***TP7 Example***

Transactions: Definition

*A transaction is **a unit of work** (a series of data manipulation) that you want to treat as "**a whole**". It has to either happen in full or not at all, and leaving the database in a consistent state.*

*In modern databases transactions also do some other things - like ensure that you can't access data that another person has written **halfway**. But the basic idea is the same - transactions are there to ensure, that no matter what happens, the data you work with will be in a **sensible state**.*

Transactions: Facts

Nature of transactions:

1. It is accomplished using DML statements (insert, delete and update)

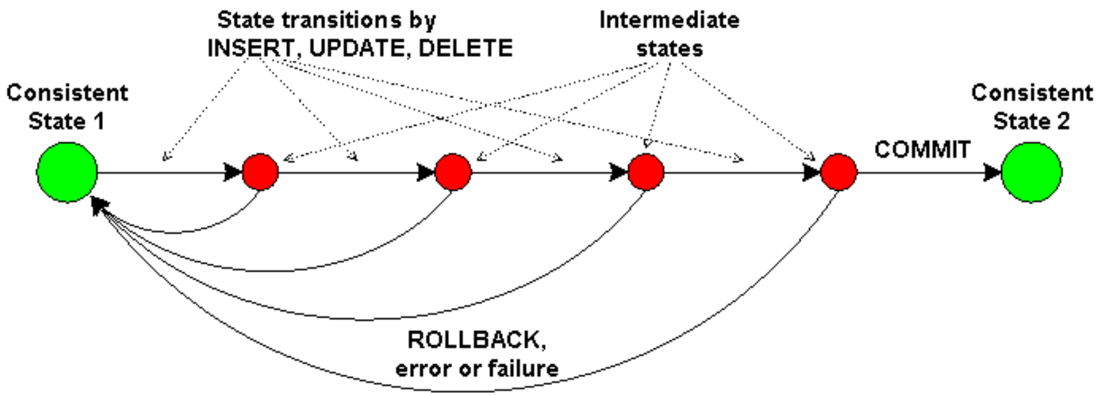
2. It always has a beginning and an end.

3. It can always save or undo.

4. When a transaction fails in the middle, no part of the transaction can be saved on the database

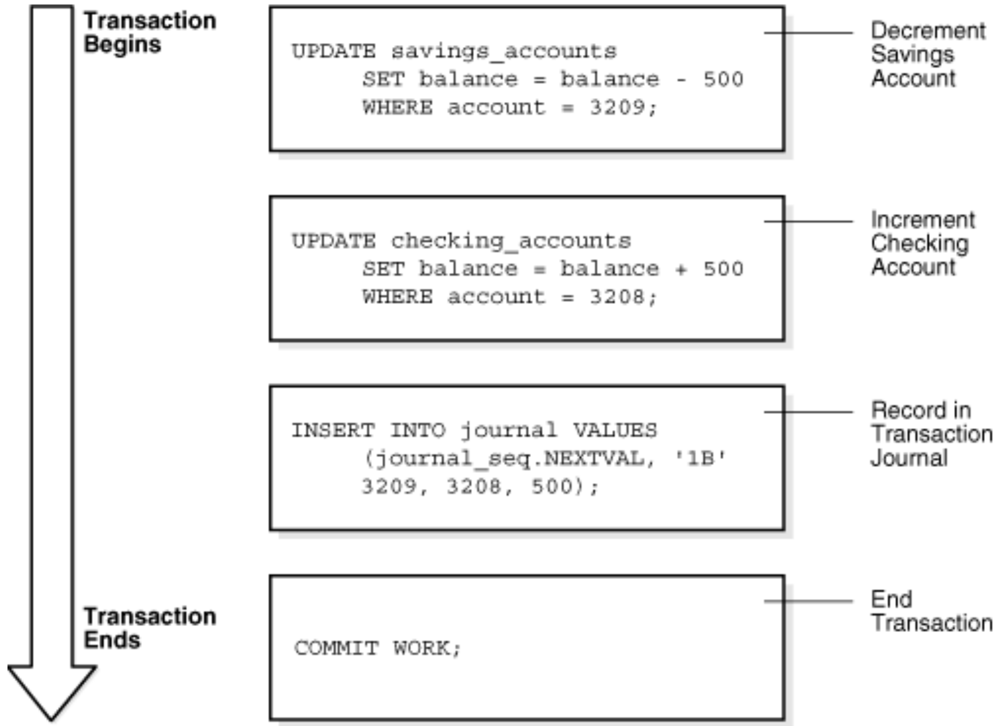
5. In oracle we had 3 commands:

- COMMIT
- ROLLBACK
- SAVEPOINT



Transactions: Classical Example

A classical example is ***transferring money*** *from one bank account to another. To do that you have first to* ***withdraw*** *the amount from the source account, and then* ***deposit*** *it to the destination account. The operation has to succeed in full. If you stop halfway, the money will be lost…!*



Transactions: Example

Customers table:

Name	Address	City	State	Zip	Customer_ID
Bill Turner	725 N. Deal Parkway	Washington	DC	20085	1
John Keith	1220 Via De Luna Dr.	Jacksonville	FL	33581	2
Mary Rosenberg	482 Wannamaker Avenue	Williamsburg	VA	23478	3
David Blanken	405 N. Davis Highway	Greenville	SC	29652	4
Rebecca Little	7753 Woods Lane	Houston	TX	38764	5

Balances table:

Average_Bal	Curr_Bal	Account_ID
1298.53	854.22	1
5427.22	6015.96	2
211.25	190.01	3
73.79	25.87	4
1285.90	1473.75	5
1234.56	1543.67	6
345.25	348.03	7

`SELECT` operation to retrieve data for Bill Turner

```
NAME:  Bill Turner
ADDRESS:  725 N. Deal Parkway
CITY:  Washington
STATE:  DC
ZIP:  20085
CUSTOMER_ID:  1
```

However, if you had used a transaction, this data change could have been detected, and all your other operations could have been rolled back.

While this information is being retrieved, **another user** with a connection to this database updates Bill Turner's address information:

```
UPDATE CUSTOMERS SET Address = "11741 Kingstowne Road"
     WHERE Name =  "Bill Turner";
```

Transactions: Implementation

Beginning a Transaction:

Syntax:

```
SET TRANSACTION {READ ONLY | USE ROLLBACK SEGMENT segment}
```

This option tells Oracle which database segment to use for rollback storage space.

Example:

```
SET TRANSACTION READ ONLY;
SELECT * FROM employees
   WHERE last_name = 'Chen';

---Do Other Operations---

COMMIT;
```

We can effectively lock a set of records until the transaction ends.

Script output:

```
Transaction READ succeeded.

EMPLOYEE_ID FIRST_NAME           LAST_NAME                 EMAIL                     PHONE_NUMBER         HIRE_DAT JOB_ID         SALARY COMMISSION_PCT MANAGER_ID DEPARTMENT_ID
----------- -------------------- ------------------------- ------------------------- -------------------- -------- ---------- ---------- -------------- ---------- -------------
        110 John                 Chen                      JCHEN                     515.124.4269         28.09.97 FI_ACCOUNT       8200                       108           100

Commit complete.
```

So, what's gonna happen if you run just this statement…?

```
SELECT * FROM employees
WHERE last_name = 'Chen';
```

Simple output:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	110	John	Chen	JCHEN	515.124.4269	28.09.97	FI_ACCOUNT	8200	(null)	108	100

Transactions: Implementation

Example:

```
SET TRANSACTION;
INSERT INTO employees VALUES
     ('110','John','Chen','JCHEN','515.124.4269','28.09.97','FI_ACCOUNT',8200,0.4,108,100);
COMMIT;
SELECT * FROM employees;
```

Transactions: Implementation

In Oracle all individual DML statements are atomic, but how can we make a group of statements to be treated as a single atomic transaction?

```
BEGIN
  SAVEPOINT first_savepoint;
  INSERT INTO .... ;
  UPDATE .... ;
  BEGIN ... END; -- some other work
  UPDATE .... ;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK TO first_savepoint;
    RAISE;
END;
```

Transactions: Example

Example:

```
-------------------transaction with procedure example ----------------------------
-- We have multiple insert and update statements like:
CREATE OR REPLACE PROCEDURE Save_Point_Test
AS
BEGIN
  INSERT INTO films(film_id, title, year, language, genre, director_id,main_actor_id,companyid)
  VALUES (10,'Fight Club',TO_DATE('1999','YYYY'),'English','Drama',10,5,3 );

  INSERT INTO main_actors(main_actor_id,first_name, last_name,birthday,country)VALUES
  (10,'David','Fincher',TO_DATE('1962','YYYY'),'USA');

  UPDATE films set title='Spider-man' WHERE film_id=2;
END;
```

This is a procedure to store some data

If we get any error in any statement, we want to rollback all insert and update statements. How can we implement commit and rollback in this stored procedure?

a new version

```
CREATE OR REPLACE PROCEDURE Save_Point_Test
AS
BEGIN

  -- We create a savepoint here.
  SAVEPOINT sp_test;

INSERT INTO films(film_id, title, year, language, genre, director_id,main_actor_id,company_id)
  VALUES (10,'Fight Club',TO_DATE('1999','YYYY'),'English','Drama',10,5,3 );

  INSERT INTO main_actors(main_actor_id,first_name, last_name,birthday,country)VALUES
  (10,'David','Fincher',TO_DATE('1962','YYYY'),'USA');

  UPDATE films set title='Spider-man' WHERE film_id=2;

-- If any exception occurs
EXCEPTION
  WHEN OTHERS THEN
    -- We roll back to the savepoint.

    ROLLBACK TO sp_test;
    -- And of course we raise again,
    -- since we don't want to hide the error.
    -- Not raising here is an error!
    RAISE;
END;
```

```
SAVEPOINT SAVEPOINT_NAME;
```

```
ROLLBACK TO SAVEPOINT_NAME;
```

This construct just guarantees that either all of the inserts and the update are done, or none of them is

Transactions: Properties

Transactions ideally have four properties, commonly known as ***ACID****:*

- ***Atomic:*** *if the change is committed, it happens in <u>one</u> fell swoop; you can never see "half a change"*
- ***Consistent:*** *the change can only happen if the new <u>state</u> of the system will be valid; any attempt to commit an invalid change will fail, leaving the system in its previous valid state*
- ***Isolated:*** *<u>no-one else</u> sees any part of the transaction until it has committed.*
- ***Durable:*** *once the change has happened - if the system says the transaction has been committed, the client does not need to worry about "flushing" the system to make the change "stick".*

check lecture slides: "Lecture1–Transactions", page 17

Synchronization of Transactions

(allowing concurrency)

What:

Instead of insisting on a strict serial transaction execution (process complete T_1, then T_2, then T_3 etc.), we want to use **multiple** user transactions in a same time.

Advantages:

- Increase the <u>throughput</u> of the system.
- <u>Minimize response time</u> for each transaction.

Transaction Schedules in DB

A **schedule** is a series of operations from **one or more transactions**. A schedule can be of two types:

- **Serial Schedule:** When one transaction completely executes before starting another transaction, the schedule is called serial schedule.
- **Concurrent Schedule:** When operations of a transaction are interleaved with operations of other transactions of a schedule, the schedule is called concurrent schedule.

Problems of Transaction Schedules

Problems that can occur for certain transaction schedules without appropriate concurrency control mechanisms:

Lost update

time	transaction T1	transaction T2
1	**read** (x)	
2	x:=x+200	
3		**read** (x)
4		x:=x+100
5	**write** (x)	
6		**write** (x)
7		**commit**
8	**commit**	

(The update performed by T1 gets lost)
T2 cannot read A while A is modified by T1

Dirty read

time	transaction T1	transaction T2
1	**read** (x)	
2	x:=x+200	
3	**write** (x)	
4		**read** (x)
5		**write** (x)
6	**rollback**	

The transaction T1 fails for some reason during the execution.
Another transaction T2 has access to the object in the meantime.
In this case, T2 read the data, which has never existed.

Inconsistent Analysis (Incorrect Summary Problem)

time	transaction T1	transaction T2
1	**read** (x)	
2		**read** (x)
3		x := x-100
4		**write** (x)
5		**read** (z)
6		z := z+ x
7		**write** (z)
8		**commit**
9	**read** (y)	
10	**read** (z)	
11	sum := x+y+z	
12	**commit**	

In this schedule, the total computed by T1 is wrong

check lecture slides: "Lecture1-Transactions", page 23

Serializability

Conflicting operations: Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions.
- They operate on the same data item.
- At Least one of them is a write operation.

Conflicts between operations of two transactions:

transaction Ti	transaction Tj
read (x)	
	read (x)

(order does not matter)

transaction Ti	transaction Tj
read (x)	
	write (x)

(order matters)

transaction Ti	transaction Tj
write (x)	
	read (x)

(order matters)

transaction Ti	transaction Tj
write (x)	
	write (x)

(order matters)

Serializability

Let's say we have schedule S, and we can reorder the instructions in it, and create two more schedules S1 and S2.

Conflict Equivalent: Refers to the schedules S1 and S2 where they **maintain the ordering of the conflicting instructions** in both of the schedules.

For example, if T1 has to read x before T2 writes x in S1, then it should be the same in S2 also. (Ordering should be maintained only for the conflicting operations).

Conflict Serializability: S is said to be conflict serializable if it is conflict equivalent to a **serial schedule** (i.e., where the transactions are executed one after the other).

A schedule is called **“correct”** if we can find a serial schedule that is conflict equivalent to it (a serializable schedule).

Serializability

Example

Question: *Consider the following schedules S1, S2, involving two transactions T1, T2. Which one of the following statement is true?*

`S1: R1(X) R1(Y) R2(X) R2(Y) W2(Y) W1(X)`

`S2: R1(X) R2(X) R2(Y) W2(Y) R1(Y) W1(X)`

a. *Both S1 and S2 are conflict serializable*

b. *Only S1 is conflict serializable*

c. *Only S2 is conflict serializable*

d. *None*

Solution in next page →

Serializability

Example

S1: R1(X) R1(Y) R2(X) R2(Y) W2(Y) W1(X)
S2: R1(X) R2(X) R2(Y) W2(Y) R1(Y) W1(X)

Solution: Two transactions of given schedules are:

```
T1: R1(X) R1(Y) W1(X)
T2: R2(X) R2(Y) W2(Y)
```

Let 's first check serializability of S1:

S1: R1(X) R1(Y) R2(X) R2(Y) W2(Y) W1(X)

- To convert it to a serial schedule, we have to swap non-conflicting operations so that S1 becomes equivalent to serial schedule T1—>T2 or T2—>T1.

- In this case, to convert it to a serial schedule, we must have to swap R2(X) and W1(X) but they are conflicting. So S1 can't be converted to a serial schedule.

S1': R1(X) R1(Y) W1(X) R2(Y) W2(Y) R2(X)

Now, we have to check S2…

Serializability

T1: R1(X) R1(Y) W1(X)
T2: R2(X) R2(Y) W2(Y)

Now, let us check serializability of S2:

S2: <u>R1(X)</u> <u>R2(X)</u> R2(Y) W2(Y) R1(Y) W1(X)

Swapping non conflicting operations R1(X) and R2(X) of S2, we get

S2’: R2(X) <u>R1(X)</u> <u>R2(Y)</u> W2(Y) R1(Y) W1(X)

Again, swapping non conflicting operations R1(X) and R2(Y) of S2’, we get

S2’’: R2(X) R2(Y) <u>R1(X)</u> <u>W2(Y)</u> R1(Y) W1(X)

Again, swapping non conflicting operations R1(X) and W2(Y) of S2’’, we get

S2’’’: R2(X) R2(Y) W2(Y) R1(X) R1(Y) W1(X)

which is equivalent to a serial schedule T2->T1.

So, correct option is **C**. Only S2 is conflict serializable.

Conflict Graph

(*precedence graph* or *serializability graph*)

It checks the serializability.

Conflict Graph

A *conflict graph*, also named *precedence graph* or *serializability graph*, is a method of concurrency control in databases to describes **dependencies** among concurrent transactions.

The conflict graph for a schedule S contains:

- A node for each committed transaction in S
- An arc from T_i to T_j if an action of T_i conflicts with one of T_j's actions.

If the graph has no cycle, then the transactions are serializable.

Conflict Graph

To draw one:

1) Draw a node for each transaction in the schedule

2) For each pair of following ordered conflict operations in S, create a directional arc in the same order

For the conflicting pair Wi(X) Rj(X), where Wi(X) happens before Rj(X).

For the conflicting pair Ri(X) Wj(X), where Ri(X) happens before Wj(X).

For the conflicting pair Wi(X) Wj(X), where Wi(X) happens before Wj(X).

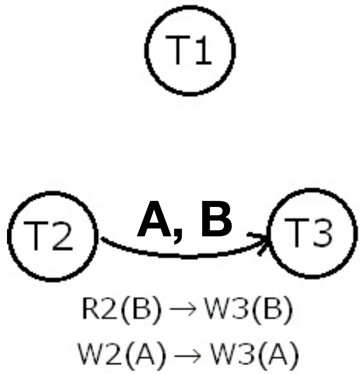
Ti	Tj	
W(X)	R(X)	– – create arc Ti -> Tj
R(X)	W(X)	– – create arc Ti -> Tj
W(X)	W(X)	– – create arc Ti -> Tj

Conflict Graph

Example

Example 1:

$$D = \begin{array}{c} \begin{matrix} T1 & T2 & T3 \end{matrix} \\ \begin{bmatrix} & R(B) & \\ R(C) & W(A) & \\ W(C) & & \\ R(D) & & \\ & & W(B) \\ W(D) & & W(A) \end{bmatrix} \end{array}$$



no cycle

a serializable schedule

S: R2(B) W2(A) R1(C) W1(C) R1(D) W3(B) W3(A) W1(D)

S': R2(B) W2(A) R1(C) W1(C) R1(D) W3(B) W1(D) W3(A)

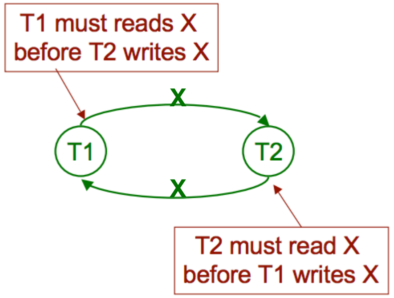
S'': R2(B) W2(A) R1(C) W1(C) R1(D) W1(D) W3(B) W3(A)

https://en.wikipedia.org/wiki/Precedence_graph

Conflict Graph

Example

Example 2:



Lost update schedule:

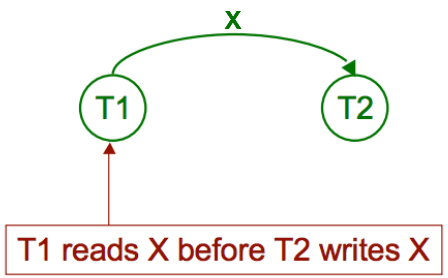
T1	T2
`Read(X)`	
`X = X - 5`	
	`Read(X)`
	`X = X + 5`
`Write(X)`	
	`Write(X)`
`COMMIT`	
	`COMMIT`

Since the graph is cyclic, we can conclude that it is **not conflict serializable** to any schedule serial schedule.

Conflict Graph

Example

Example 3:



T1	T2
Read(X)	
X = X – 5	
Write(X)	
	Read(X)
	X = X + 5
	Write(X)
COMMIT	
	COMMIT

Conflict Graph

Example

Example 4:

Schedule:

S_1 : $r_2(A)\ r_1(B)\ w_2(A)\ r_3(A)\ w_1(B)\ w_3(A)\ r_2(B)\ w_2(B)$

Make it easier by grouping the action by their *different* DB elements:

$r_2(A)\quad w_2(A)\ r_3(A)\quad w_3(A)$ ← A

$r_1(B)\quad w_1(B)\quad r_2(B)\ w_2(B)$ ← B

Now you can see the precedence relationships *easier* (find all *conflicting* operations):

$r_2(A)\quad w_2(A)\ r_3(A)\quad w_3(A)$

$r_1(B)\quad w_1(B)\quad r_2(B)\ w_2(B)$

Graph with *all* precedence relationships:

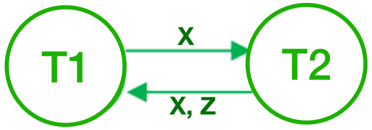
$$T_1 \xrightarrow{B} T_2 \xrightarrow{A} T_3$$

Conflict Graph

Example

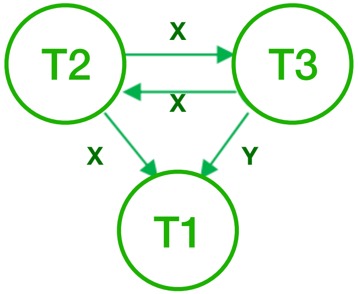
Example 5:

r2(X) r1(X) r2(Z) w2(Z) w2(X) r1(Z) w1(X)



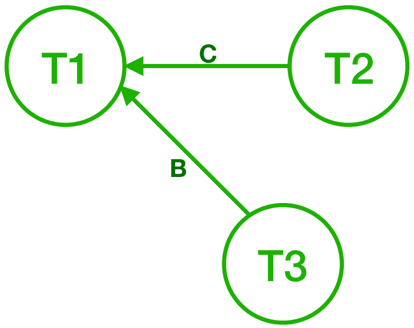
Example 6:

r2(X) r3(Y) w3(X) r1(Y) w2(X) w1(Y) w1(X)



Example 7:

W2(C) R1(A) W3(B) R1(C) R3(B) R3(A) W1(B)



Recoverability of Schedules

- As discussed, a transaction may not execute completely due to hardware failure, system crash or software issues.

- In that case, we have to rollback the failed transaction.

- But some other transaction may also have used values produced by failed transaction. So we have to rollback those transactions as well.

Recoverability of Schedules

Example

Dirty Read problem

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		
Failure Point				
Commit;				

So this schedule is irrecoverable.

This table shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. But we have already committed that.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
Failure Point				
Commit;				
		Commit;		

✅ **This schedule is recoverable.**

https://www.geeksforgeeks.org/dbms-recoverability-of-schedules/

Recoverability of Schedules

Example

Example 2:

```
S1: R1(x), W1(x), R2(x), R1(y), R2(y), W2(x), W1(y), C1, C2;

T1: R1(x), W1(x),      , R1(y),      ,      , W1(y), C1,   ;
T2:      ,      , R2(x),      , R2(y), W2(x),      ,   , C2;
```

✅ **So this schedule is a recoverable schedule.**

Example 3:

```
S2: R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), W3(y), R2(y), W2(z), W2(y), C1, C2, C3;

T1: R1(x),      , R1(z),      ,      , W1(x),      ,      ,      ,      , C1,   ,   ;
T2:      , R2(x),      ,      ,      ,      ,      , R2(y), W2(z), W2(y),   , C2,   ;
T3:      ,      ,      , R3(x), R3(y),      , W3(y),      ,      ,      ,   ,   , C3;
```

So this schedule is an irrecoverable schedule.

https://www.geeksforgeeks.org/dbms-recoverability-of-schedules/

Recoverability of Schedules

Question: Which of the following scenarios may lead to an **irrecoverable error** in a database system?

(A) A transaction writes a data item after it is read by an uncommitted transaction.

(B) A transaction reads a data item after it is read by an uncommitted transaction.

(C) A transaction reads a data item after it is written by a committed transaction.

(D) A transaction reads a data item after it is written by an uncommitted transaction.

https://www.geeksforgeeks.org/dbms-recoverability-of-schedules/

How do we ensure Serializability?

Locks in Database Management Systems

Locking *enforces serializability by ensuring that no two* <u>*transactions*</u> *access* <u>*conflicting*</u> *objects in an "incorrect"* <u>*order*</u>*.*

Concurrency Control: Locks

Types of locks that can be used in a transaction T:

- **slock(X) or rl(X):** shared-lock (read-lock); no other transaction than T can write data object X, but they can read X

- **xlock(X) or wl(X):** exclusive-lock (write-lock); T can read/write data object X; no other transaction can read/write X

- **unlock(X):** unlock data object X

Concurrency Control: Locks

- **Compatibility matrix** of locks

existing lock (columns); *requested lock* (rows)

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

- General locking algorithm
  1. Before using a data item x, transaction requests lock for x from the lock manager
  2. If x is already locked and the existing lock is incompatible with the requested lock, the transaction is delayed
  3. Otherwise, the lock is granted

Exclusive lock vs. Shared lock

(xl/write-lock/wl) *(sl/read-lock/rl)*

Think of a **blackboard** (lockable) in a class containing a teacher (writer) and some students (readers).

While a teacher is writing something **(exclusive lock)** on the board:

1. Nobody can read it, because it's still being written, and the teacher is blocking the view
   - **If an object is exclusively locked, shared locks cannot be obtained.**

2. Other teachers can not come and write
   - **If an object is exclusively locked, other exclusive locks cannot be obtained.**

When the students are reading **(shared locks)** what is on the board:

1. They all can read what is on it, together
   - **Multiple shared locks can co-exist.**

2. The teacher waits for them to finish reading before she clears the board to write more
   - **If one or more shared locks already exist, exclusive locks cannot be obtained.**

Locks Compatibility

Example

Example 1:

T1	T2
RL(A)	
R(A)	
RL(B)	
RL(B)	
	RL(A)
	RL(A)
	UL(A)
UL(A)	
UL(B)	

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

- In this example, read locks for both T1 and T2 over A were acquired.
- Since both transactions did nothing but read, this is easily identifiable as a serializable schedule.

Locks Compatibility

Example

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

Example 2:

T1	T2
WL(A)	
R(A)	
	RL(A)
W(A)	
UL(A)	
	RL(A)
	RL(A)
	UL(A)
commit	
	commit

`Attempt to RL(A):` **`fails`**

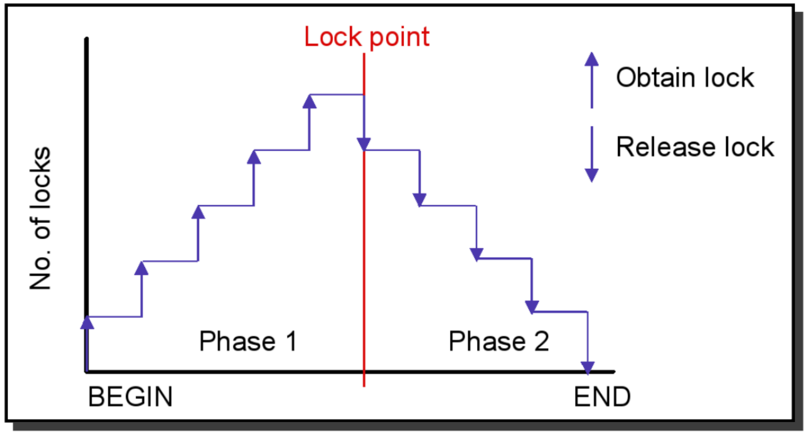
Conflict

It's not allowed a lock to be granted until the write lock had been freed.

Two Types of Lock Protocols

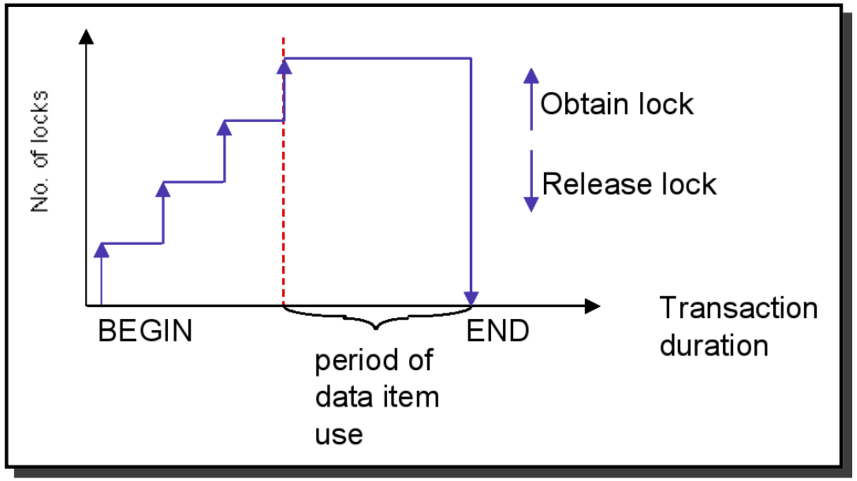
2PL vs. Strict-2PL

- **Two-phase locking** protocol
  - Each transaction is executed in two phases
    - ∗ **Growing phase:** the transaction obtains locks
    - ∗ **Shrinking phase:** the transaction releases locks
  - The **lock point** is the moment when transitioning from the growing phase to the shrinking phase



- **Strict 2PL locking** protocol
  - Holds the locks till the end of the transaction
  - Cascading aborts are avoided

in 1 transaction



Two Phase Locking (2PL)

to ensure serializability

- Each object has associated with it a lock.

- An appropriate lock must be acquired before a transaction (txn) accesses the object.

- Two locks, $pl_i[x]$ and $ql_j[y]$, conflict if $x=y$ and $i \ne j$; and p and q are conflicting operations.

2PL is Defined By 3 rules

1. To grant a lock, the scheduler <u>checks</u> if a conflicting lock has already been assigned, if so, delay, otherwise set lock and grant it.

2. A lock cannot be <u>released</u> at least until the DM acknowledges that the operation has been performed.

3. Once the scheduler releases a lock for a txn, it may **not** subsequently acquire any more locks (on any item) for that txn.

2-phase Locking Protocol

Example

Example:

T1 is two-phased - all of its locks are acquired before it releases any of them

T2 is not - it releases its lock on X and then goes on to later acquire a lock on Y

T1	T2
S-lock(X)	S-lock(X)
Read(X)	Read(X)
X-lock(Y)	Unlock(X)
Unlock(X)	X-lock(Y)
Read(Y)	Read(Y)
Y = Y + X	Y = Y + X
Write(Y)	Write(Y)
Unlock(Y)	Unlock(Y)

2-phase Locking Protocol

Example

Example: Consider the following two transactions:

T1

```
Read ( x )
x ← x + 1
Write ( x )
Read ( y )
y ← y − 1
Write ( y )
```

T2

```
Read ( x )
x ← x ∗ 2
Write ( x )
Read ( y )
y ← y ∗ 2
Write ( y )
```

```
S1 = {wl1(x), R1(x), W1(x), ul1(x)
      wl2(x), R2(x), W2(x), ul2(x)
      wl2(y), R2(y), W2(y), ul2(y)
      wl1(y), R1(y), W1(y), ul1(y) }
```

S1 is not valid schedule in 2PL protocol (e.g., after ulr1(x) in line 1, transaction T1 cannot request the lock wl1(y) in line 4).

- Is S1 serializable? **NO**

why?

- S1 cannot be transformed into a serial schedule by using only non-conflicting swaps
- The result is different from the result of any serial execution

Solution in next page

2-phase Locking Protocol

Example

```
S1 = {wl1(x), R1(x), W1(x), ul1(x)
      wl2(x), R2(x), W2(x), ul2(x)
      wl2(y), R2(y), W2(y), ul2(y)
      wl1(y), R1(y), W1(y), ul1(y) }
```

```
S2 = {wl1(x), R1(x), W1(x),
      wl1(y), R1(y), W1(y), ul1(x), ul1(y)
      wl2(x), R2(x), W2(x),
      wl2(y), R2(y), W2(y), ul2(x), ul2(y) }
```

S2 is valid schedule in the 2PL protocol

T1

```
Read ( x )
x ← x + 1
Write ( x )
Read ( y )
y ← y − 1
Write ( y )
```

T2

```
Read ( x )
x ← x ∗ 2
Write ( x )
Read ( y )
y ← y ∗ 2
Write ( y )
```

Example:

2-phase Locking Protocol

Example

S1

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

The schedule is conflict serializable.

S1

Schedule

T1	T2
RL(A)	
R(A)	
	RL(A)
	R(A)
WL(B)	
R(B)	
W(B)	
UL(A)	
UL(B)	
commit	
	WL(B)
	R(B)
	W(B)
	UL(A)
	UL(B)
	commit

Example:

2-phase Locking Protocol

Example

S1

T1	T2	
WL(A)		
R(A)		
W(A)		
	RL(A)	not allowed
WL(B)		
R(B)		
W(B)		
UL(A)		
UL(B)		
	WL(A)	
	R(A)	
	W(A)	
	WL(B)	
	R(B)	
	W(B)	
	UL(A)	
	UL(B)	
commit		
	commit	

S2

T1	T2	
WL(A)		
R(A)		
W(A)		
WL(B)		
R(B)		
W(B)		
UL(A)		
UL(B)		
	RL(A)	not allowed
	WL(A)	
	R(A)	
	W(A)	
	WL(B)	
	R(B)	
	W(B)	
	UL(A)	
	UL(B)	
commit		
	commit	

S3

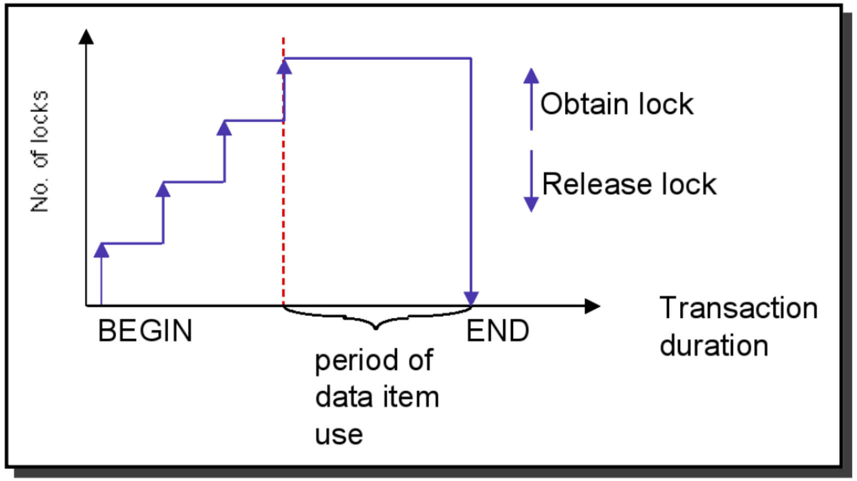
T1	T2
WL(A)	
R(A)	
W(A)	
WL(B)	
R(B)	
W(B)	
UL(A)	
UL(B)	
	WL(A)
	R(A)
	W(A)
	WL(B)
	R(B)
	W(B)
	UL(A)
	UL(B)
commit	
	commit

Strict Two-phase Locking

(Strict 2PL) Protocol

- Each transaction must obtain a S (shared) lock on object before reading, and an X-lock (exclusive) on object before writing.
- All locks held by a transaction are released when the transaction completes.
- If a transaction holds an X-lock on an object, no other transactions can get a lock (S or X) on that object.

in 1 transaction



Strict Two-phase Locking

(Strict 2PL) Protocol

Example:

Which of these are legal schedules (that interleave transactions T1 and T2) under strict-2PL?

These schedules only show the locks (X: exclusive lock; S: shared lock), reads (R) and writes (W).

s1

T1	T2
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

s2

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
X(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

s3

T1	T2
X(A)	
R(A)	
W(A)	
	X(A)
	R(A)
	W(A)
X(B)	
R(B)	
W(B)	
Commit	
	X(B)
	R(B)
	W(B)
	Commit

Strict Two-phase Locking

(Strict 2PL) Protocol

Example:

s1

T1	T2
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

OK

s2

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
X(A)	**not allowed**
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

NO

s3

T1	T2
X(A)	**not allowed**
R(A)	
W(A)	
	X(A)
	R(A)
	W(A)
X(B)	
R(B)	
W(B)	
Commit	
	X(B)
	R(B)
	W(B)
	Commit

NO

TP7 - Examples

Example: Consider the following schedule:

T_1	T_2
read(A)	
	write(B)
read(B)	

i. Is this schedule possible under the two-phase locking protocol?

ii. If yes, add lock and unlock instructions.

TP7 - Examples

i. Yes, it is possible under the two-phase locking protocol.

ii.

	T_1	T_2
1	lock-S(A)	
2	read(A)	
3		lock-X(B)
4		write(B)
5		unlock(B)
6	lock-S(B)	
7	read(B)	
8	unlock(A)	
9	unlock(B)	

Thank you!