TRANSACTION MANAGEMENT

Transaction Processing

- Consider large databases and hundreds of concurrent users working on database.
- Examples of such systems are airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications.
- Concurrency means that many users can access data at the same time
- If concurrency is not allowed in large databse systems they suffer from performance issues.

Transaction Processing

- A transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit.
- The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database).
- Why Concurrency Control Is Needed?
  - Several problems can occur when concurrent transactions execute in an uncontrolled manner.
    - The Lost Update Problem
    - The Temporary Update (or Dirty Read) Problem
    - The Incorrect Summary Problem
    - The Unrepeatable Read Problem.

Transaction Example

- A transaction begins with the first executable SQL statement.
- A transaction ends when it is committed or rolled back, either explicitly with a COMMIT or ROLLBACK statement or implicitly when a DDL statement is issued.
- To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:
  - Decrement the savings account
  - Increment the checking account
  - Record the transaction in the transaction journal

Transaction Example

```
UPDATE Savings_Accounts SET balence = balence - 500
WHERE account = 3209;

UPDATE Checking_Accounts SET balence = balence + 500
WHERE account = 3208;

Insert into journal values(journal_seq.NEXTVAL, 'IB',3209,3208,500);

COMMIT;
```

Transaction Control Language(TCL)

- TCL Commands
  - COMMIT
  - ROLLBACK
  - SAVEPOINT
  - ROLLBACK TO SAVEPOINT
  - SET TRANSACTION

Transaction Control Language(TCL)

- COMMIT
  - A COMMIT statement ends a transaction and makes all changes visible to other users.
  - Permantly save transaction in databse
- Syntax: COMMIT;

Transaction Control Language(TCL)

- ROLLBACK
  - A ROLLBACK statement undoes all work performed since the transaction began
- Syntax: ROLLBACK;

Transaction Control Language(TCL)

- SAVEPOINT
  - A savepoint is a way of implementing subtransactions by indicating a point within a transaction that can be "rolled back to" without affecting any work done in the transaction before the savepoint was created.
  - Multiple savepoints can exist within a single transaction
- Syntax: SAVEPOINT <name>;

Transaction Control Language(TCL)

- ROLLBACK TO SAVEPOINT
  - A ROLLBACK statement undoes all work performed upto savepoint began
- Syntax: ROLLBACK TO SAVEPOINT <name>;

Transaction Processing

- A transaction is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit.
- The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database).
- Why Concurrency Control Is Needed?
  - Several problems can occur when concurrent transactions execute in an uncontrolled manner.
    - The Lost Update Problem
    - The Temporary Update (or Dirty Read) Problem
    - The Incorrect Summary Problem
    - The Unrepeatable Read Problem.

The Lost Update problem

- A Simplified airline reservations database
- Each record includes the number of reserved seats among other information.
- a) shows a transaction T1 that transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y.
- b) shows a simpler transaction T2 that just reserves M seats on the first flight (X) referenced in transaction T1

(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);

The Lost Update problem

- DB writer process will dose not write to database files because of issuing commit statement.
- Commit just means that this is a end of transaction.
- It has its own scenarios like when
  - checkpoint happens
  - or dirty buffers reaches threshold
  - or there is no free buffer etc.

(a)

T_1

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

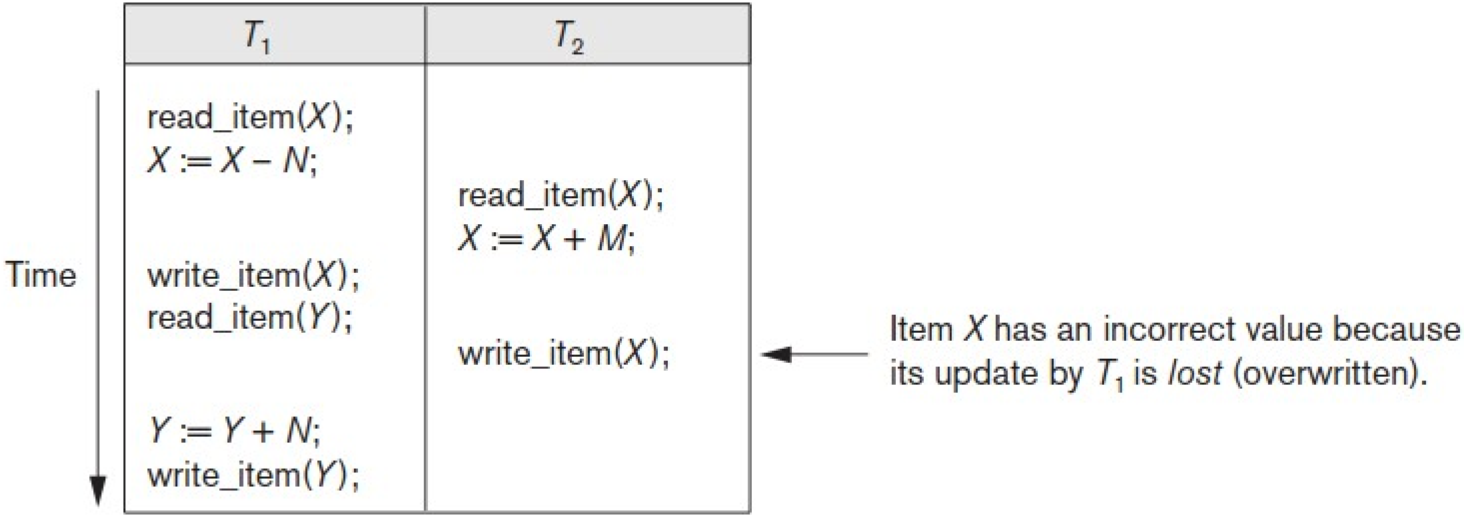
(b)

T_2

read_item(X);
$X := X + M$;
write_item(X);

The Lost Update problem

- Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown below



- then the final value of item X is incorrect because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.

The Lost Update problem

- Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown below

$$Assume\ X=80\,,N=5$$
$$X=X-N=80-5=75$$
$$Y=Y+N$$

Time

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

$$X=80\,,M=4$$
$$X=X+M=80+4=84$$

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

$$X=X+M=75+4=97$$

- then the final value of item X is incorrect because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.

The Dirty Read problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

The Dirty Read problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.

$X = 80, N = 5$

$X = 80 - 5 = 75$

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y); *Rollback*;	

$X = 75, M = 4$

$X = 75 + 4 = 79$

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

The Unrepeatable Read Problem.

- Another problem that may occur is called unrepeatable read, where a transaction T1 reads the same item twice and the item is changed by another transaction T2 between the two reads.
- Hence, T1 receives different values for its two reads of the same item.
- This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights.
- When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item

The Unrepeatable Read Problem.

	T_1	T_2
$Assume\ Avilable\ Balence(X) = 10000$	$Read(X)$	
	$\cdot$	$Read(X)$
	$\cdot$	$X = X - 8500$
$Avilable\ Balence(X) = 1500$	$Read(X)$	
	$X = X - N$	

The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

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).

The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

$Assume\ X=80$

$N=5, Y=50$

$X=80-5=75$

$Y=Y+N$

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).

$Assume\ A=20, total=0$

$total=0+20=20$

.

.

$total=20+75$

$total=95+50=145$

$total=20+80+50=150$

Concurency Control

- Concurrency Control goal
  - Avoid inconsistency
  - Databse user execute statements without worrying about what other user are doing in databse.
- Simple Solution
  - Execute statements in isolation(no concurrency)
  - But it is not possible in large databases because performance will be slow
- Solution
  - Execute statements with concurrency but result should be same as serial execution means with out any inconsistency problem

Types of Failures

- Failures are generally classified as transaction, system, and media failures.
- There are several possible reasons for a transaction to fail in the middle of execution:
  - A computer failure (system crash)
  - A transaction or system error(like division by Zero)
  - Local errors or exception conditions detected by the transaction. (insufficient balence in account)
  - Concurrency control enforcement.(The concurrency control method may abort a transaction because it violates serializability)
  - Disk failure and Physical problems(power or air-conditioning failure, fire, theft, sabotage)

Why Recovery from failure?

- The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not, because the whole transaction is a logical unit of database processing.
- If a transaction fails after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.
- Otherwise databse will be in inconsistent state.
- Goal of Recovery from failure
  - Guarantee all or nothing execution regardless of failure

Transaction Processing

- The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

Transaction Processing

- Transaction, which is used to represent a logical unit of database processing.
- It must be completed in its entirety to ensure correctness, independent of other transactions
- A transaction is typically implemented by a computer program that includes database commands such as retrievals, insertions, deletions, and updates

ACID Properties

- To avoid problems caused by concurency and system failures transactions should obey some essential properties
- These properties are called ACID properties

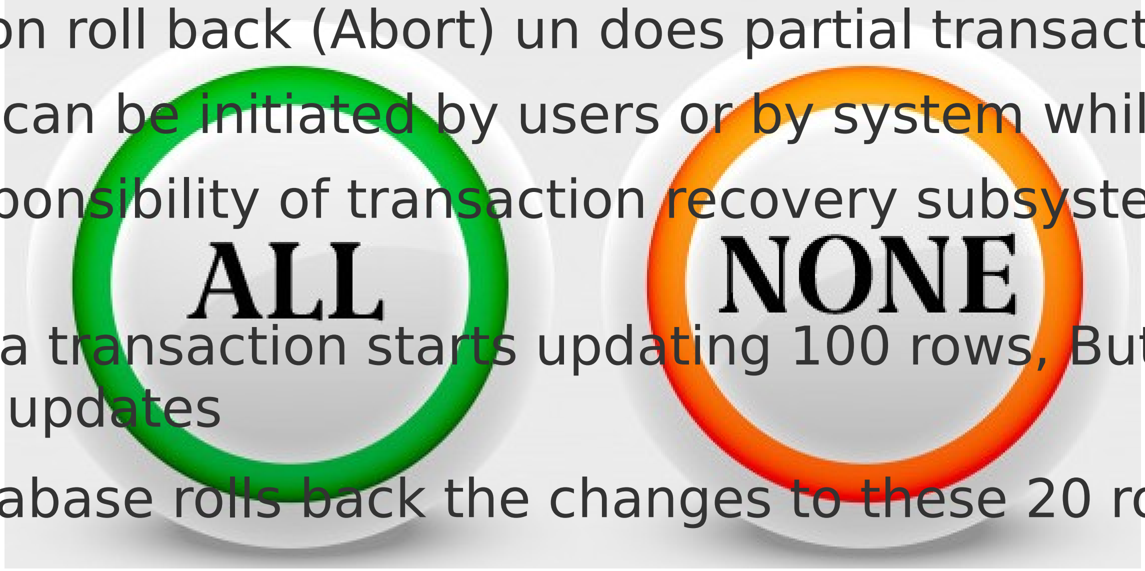
1. Atomicity
2. Consistency
3. Isolation
4. Durability

ACID Properties - Atomicity

- Atomicity means multiple operations can be grouped into a single logical entity
- Each transaction should execute in its entirity(“all or nothing”)
  - If system crashes before commit by logging mechanism when system recovering from crash partial trnsactions will be undone
  - Transaction roll back (Abort) un does partial transactions
  - Roll back can be initiated by users or by system while recovery
  - It is a responsibility of transaction recovery subsystem to achive atomicity
- **Example:** If a transaction starts updating 100 rows, But the system fails after 20 updates
- Then the database rolls back the changes to these 20 rows.

ACID Properties - Atomicity

- Atomicity means multiple operations can be grouped into a single logical entity
- Each transaction should execute in its entirity(“all or nothing”)
  - If system crashes before commit by logging mechanism when system recovering from crash partial trnsactions will be undone
  - Transaction roll back (Abort) un does partial transactions
  - Roll back can be initiated by users or by system while recovery
  - It is a responsibility of transaction recovery subsystem to achive atomicity



- **Example:** If a transaction starts updating 100 rows, But the system fails after 20 updates
- Then the database rolls back the changes to these 20 rows.

ACID Properties - Consistency

- The transaction takes the database from one consistent state to another consistent state.
- For example, in a banking transaction that debits a savings account and credits a checking account, a failure must not cause the database to credit only one account, which would lead to inconsistent data.
- In DBMS there is no component to take care of this
- Can be implemented only by integrity constraints
- If serializability holds constraints always holds

ACID Properties - Consistency

- The transaction takes the database from one consistent state to another consistent state.

$Assume\ X=80, N=5$

$X=X-N=80-5=75$

Time

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

$Y=Y+N$

$X=80, M=4$

$X=X+M=80+4=84$

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

$X=X+M=75+4=97$

ACID Properties - Consistency

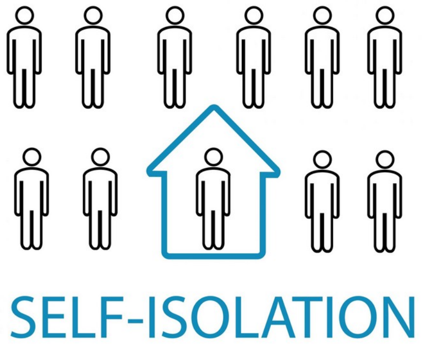
- The transaction takes the database from one consistent state to another consistent state.
- For example, in a banking transaction that debits a savings account and credits a checking account, a failure must not cause the database to credit only one account, which would lead to inconsistent data.
- In DBMS there is no component to take care of this
- Can be implemented only by integrity constraints
- If serializability holds constraints always holds

ACID Properties - Isolation

- The execution of a transaction should not be interfear with any other transaction
- The effect of a transaction is not visible to other transactions until the transaction is committed.
- For example, one user updating the Employee table does not see the changes to employees made concurrently by another user. Thus, it appears to users as if transactions are executing serially.
- Implemented by serializability by locking protocols

ACID Properties - Isolation

- The execution of a transaction should not be interfear with any other transaction
- The effect of a transaction is not visible to other transactions until the transaction is committed.
- For example, one user updating the Employee table does not see the changes to employees made concurrently by another user. Thus, it appears to users as if transactions are executing serially.
- Implemented by serializability by locking protocols



ACID Properties - Isolation

- The execution of a transaction should not be interfear with any other transaction

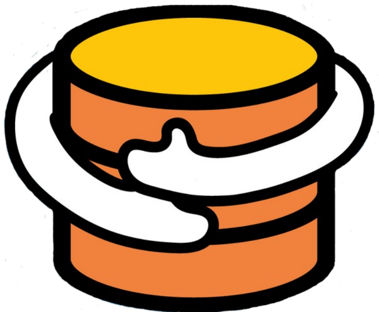
	T_1	T_2
$Assume\ Avilable\ Balence(X) = 10000$	$Read(X)$	
	$\cdot$	$Read(X)$
	$\cdot$	$X = X - 8500$
$Avilable\ Balence(X) = 1500$	$Read(X)$	
	$X = X - N$	

ACID Properties - Durability

- Once the changes applied to the databse by commited transaction it must be made perminant
- Changes made by committed transactions are permanent. After a transaction completes, If crashes all changes should not be lost
- The database ensures through its recovery mechanisms that changes from the transaction are not lost.
- It is implemented by using logging protocols with recovery system

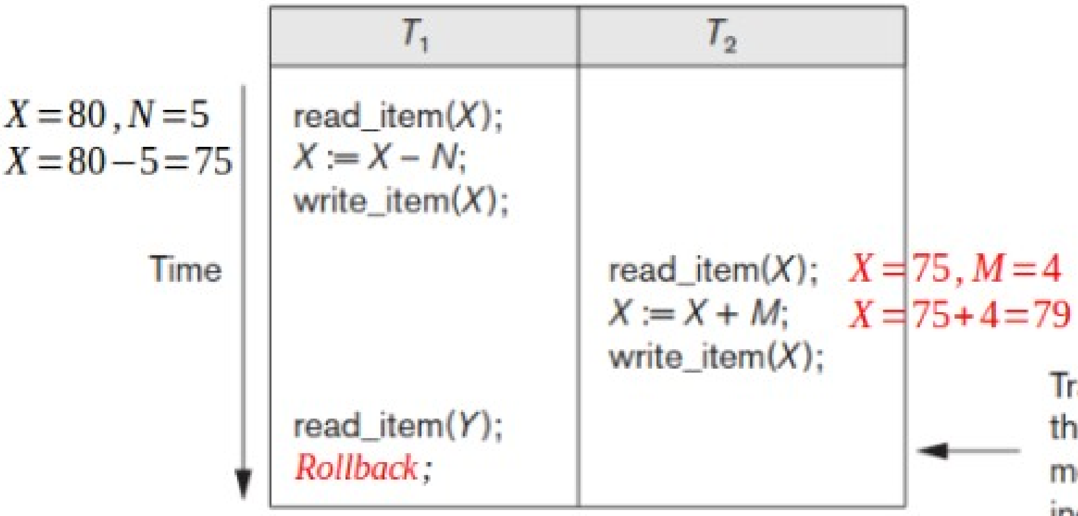
ACID Properties - Durability

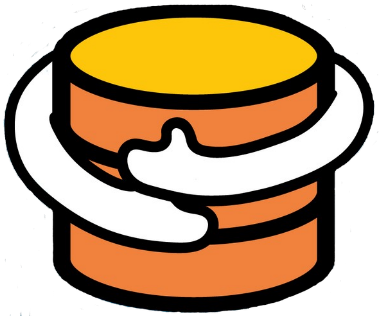
- Once the changes applied to the databse by commited transaction it must be made perminant
- Changes made by committed transactions are permanent. After a transaction completes, If crashes all changes should not be lost
- The database ensures through its recovery mechanisms that changes from the transaction are not lost.
- It is implemented by using logging protocols with recovery system



ACID Properties - Durability

- Changes made by committed transactions are permanent. After a transaction completes, If crashes all changes should not be lost





Isolation Levels

- To avoid problems caused by concurency and system failures transactions should obey ACID properties
- It is not our task to implement ACID properties. It is a task of DBMS software developers (Oracle, mysql).
- If database server support ACID properties you can use it by choose correct isolation level and use transactions

1. Read uncommited
2. Read commited
3. Repetable read
4. Serializable

Isolation level controls the extent to which the given transaction is exposed to the actions of other transactions executing currently

Isolation Levels

```sql
CREATE TABLE temp (a INT primary key, b INT);
INSERT INTO temp VALUES (1,2);
INSERT INTO temp VALUES (2,3);
INSERT INTO temp VALUES (3,2);
INSERT INTO temp VALUES (4,3);
INSERT INTO temp VALUES (5,2);
COMMIT;
```

Isolation Levels

READ UNCOMMITTED

- Read Uncommitted allows you to see uncommitted rows in another transaction
- There is no guarantee the other transaction will commit
- Dirty reads are allowed so less consistency
- Allows more concurrency

The Dirty Read problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.

$X = 80, N = 5$

$X = 80 - 5 = 75$

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y); *Rollback*;	

$X = 75, M = 4$

$X = 75 + 4 = 79$

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

Isolation Levels

READ UNCOMMITTED

- Read Uncommitted allows you to see uncommitted rows in another transaction
- There is no guarantee the other transaction will commit
- Dirty reads are allowed so less consistency
- Allows more concurrency

```
SET TRANSACTION isolation level Read Uncommitted;
```

Isolation Levels

READ COMMITTED

- Each command can view all the changes saved in the database at the time it starts(statement-level consistency)
- Any changes saved by other sessions after it starts are hidden.
- If a transaction requires row locks held by another transaction, the transaction will wait until the row locks are released.
- Dirty reads are not allowed so less concurrency than READ UNCOMMITTED but more consistency

Isolation Levels – Read Committed

```
# Transaction 1

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 5 WHERE b = 3;
```

```
# Transaction 2

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 4 WHERE b = 2;
```

Isolation Levels – Read Committed

```
# Transaction 1

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 5 WHERE b = 3;
```

```
x-lock(1,2); unlock(1,2)
x-lock(2,3); update(2,3) to (2,5); retain x-lock
x-lock(3,2); unlock(3,2)
x-lock(4,3); update(4,3) to (4,5); retain x-lock
x-lock(5,2); unlock(5,2)
```

Isolation Levels – Read Committed

```
# Transaction 2

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 4 WHERE b = 2;
```

```
x-lock(1,2); update(1,2) to (1,4); retain x-lock
x-lock(2,3); unlock(2,3)
x-lock(3,2); update(3,2) to (3,4); retain x-lock
x-lock(4,3); unlock(4,3)
x-lock(5,2); update(5,2) to (5,4); retain x-lock
```

Isolation Levels – Read Committed

```
# Transaction 1

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 5 WHERE b = 3;

select * from temp;
```

```
# Transaction 2

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 4 WHERE b = 2;

select * from temp;
```

Isolation Levels – Read Committed

```
# Transaction 1

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 5 WHERE b = 3;
commit;
```

```
# Transaction 2

SET TRANSACTION isolation level Read Committed;
UPDATE temp SET b = 4 WHERE b = 2;
select * from temp;
```

The Unrepeatable Read Problem.

	T_1	T_2
$Assume\ Avilable\ Balence(X)=10000$	$Read(X)$	
	$\cdot$	$Read(X)$
	$\cdot$	$X=X-8500$
$Avilable\ Balence(X)=1500$	$Read(X)$	
	$X=X-N$	

The Unrepeatable Read Problem.

```
# transaction1
SET transaction isolation level read commited;
select * from temp;

                                        # transaction2
                                        UPDATE temp SET b = 7 WHERE b = 5;
                                        commit;

select * from temp;
commit;
```

Phantom Read

- A phantom read is a special case of unrepeatable reads.
- This happens when another session inserts or deletes rows that match the where clause of your query.
- So repeated queries can return different rows

Phantom Read

```
# transaction1
SET transaction isolation level read commited;
select * from temp;

                                        # transaction2
                                        insert into temp values(100,200);
                                        commit;


select * from temp;
commit;
```

Isolation Levels

REPEATABLE READ

- An item read multiple times can not change value
- Implements lock-based concurrency control
- It keeps read and write locks (acquired on selected data) until the end of the transaction.
- So there is no chance of unrepeatable reads and dirty reads
- might be chance of phantom tuples(phantom reads)
- less concurrency than READ COMMITTED but more consistency

Isolation Levels

REPEATABLE READ

- The intent of repeatable read in the SQL standard is to provide consistent results from a query.
- But Oracle Database already has this in read committed!
- So it has no use for this level and does not implement it.

Isolation Levels – Repeatable Read

```
# Transaction 1

SET TRANSACTION isolation level Repeatable Read;
UPDATE temp SET b = 7 WHERE b = 5;
```

```
# Transaction 2

UPDATE temp SET b = 6 WHERE b = 4;
```

```
# Transaction 1

SET TRANSACTION isolation level Repeatable Read;
UPDATE temp SET b = 7 WHERE b = 5;

x-lock(1,4); retain x-lock
x-lock(2,5); update(2,5) to (2,7); retain x-lock
x-lock(3,4); retain x-lock
x-lock(4,5); update(4,5) to (4,7); retain x-lock
x-lock(5,4); retain x-lock


# Transaction 2

UPDATE temp SET b = 6 WHERE b = 4;

x-lock(1,4); block and wait for first UPDATE to commit or roll back
```

Isolation Levels

SERIALIZABLE

- Allows interleaving but result sholud be equal to searial schedule
- Provides transaction-level consistency
- more strict rules so it allows less concurrency, provides more consistency.
- You can only view changes committed in the database at the time your transaction starts.
- Any changes made by other transactions after this are hidden from your transaction.

Isolation Levels - Serializable

```
# transaction1
SET TRANSACTION isolation level SERIALIZABLE;
select * from temp;

                                        # transaction2
                                        insert into temp values(500,700);
                                        commit;

select * from temp;
commit;
```

Isolation Levels - Serializable

```
# transaction1
SET TRANSACTION isolation level SERIALIZABLE;
select * from temp;

                                        # transaction2
                                        insert into temp values(500,700);
                                        commit;

select * from temp;
commit;

# transaction3
select * from temp;
commit;
```

Isolation Levels vs Read phenomena

	Type of Violation		
Isolation Level	Dirty Reads	Non-repeatable Reads	Phantom Reads
Read Uncommitted	✔	✔	✔
Read Committed	✘	✔	✔
Repeatable Reads	✘	✘	✔
Serializable	✘	✘	✘

TCL – Set Transaction

- Set transaction statement can also use to set transaction access mode(read-only or both read write).
- **Read only:**
  - It works same way as serializable
  - One more restriction that you can only run selects
  - Users must also only be able to read data. You need to stop all non-select statements
  - useful in reporting environments

  Syntax: set transaction **read only**;

TCL – Set Transaction

- Set transaction statement can also use to set transaction access mode(read-only or both read write).
- **Read - write:**
  - It works same as read committed isolation level

  Syntax: set transaction **read write**;

TCL – Set Transaction

- SET TRANSACTION name
  - SET TRANSACTION can also assign transaction name.

Schedules of Transactions

- When transactions are executing concurrently in an interleaved fashion
- Then the **order of execution** of operations from all the various transactions is known as a **schedule**
- A schedule S of n transactions T_1, T_2, ... , T_n is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in the schedule S.
- However, for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i

Schedules of Transactions

- A shorthand notation for describing a schedule

Symbol	Operation
r	read_item
w	write_item
c	commit
a	abort

- and appends as a subscript the transaction id (transaction number) to each operation in the schedule.

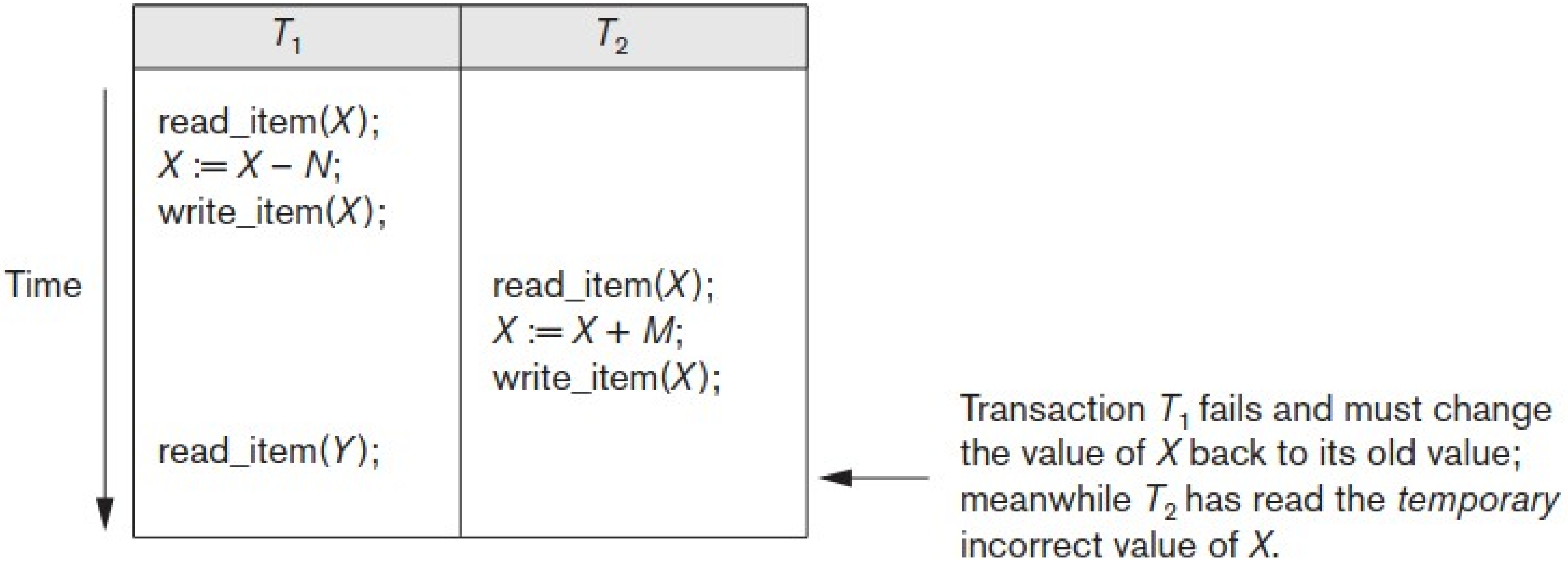
Schedules of Transactions

Time	T_1	T_2
	read_item(X); $X := X - N$;	
		read_item(X); $X := X + M$;
	write_item(X); read_item(Y);	
		write_item(X);
	$Y := Y + N$; write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

$$S_a:\ r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Schedules of Transactions



$$S_b:\ r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

Conflicting Operations in a Schedule

- Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:
  - 1. they belong to differ-ent transactions
  - 2. they access the same item X
  - 3. at least one of the operations is a write_item(X)

$$S_a:\ r_1(X);r_2(X);w_1(X);r_1(Y);w_2(X);w_1(Y);$$

- In schedule S_a, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$
- the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations

Conflicting Operations in a Schedule

- the operations $w_2(X)$ and $w_1(Y)$ do not conflict because they operate on distinct data items X and Y;
- and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction

$$S_a: \; r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Conflicting Operations in a Schedule

- Intuitively, two operations are conflicting if changing their order can result in a different outcome.
- For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, then the value of X that is read by transaction T1 changes, because in the second ordering the value of X is read by $r_1(X)$ after it is changed by $w_2(X)$, whereas in the first ordering the value is read before it is changed.
- This is called a **read-write conflict.**

$X = 100$

S_1

	T_1	T_2
$X = 100$	$r_1(X)$	
		$w_2(X, 200)$

S_2

	T_1	T_2
		$w_2(X, 200)$
$X = 200$	$r_1(X)$	

Conflicting Operations in a Schedule

- The other type is called a **write-write conflict**
- For example we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$.
- For a write-write conflict, the last value of X will differ because in one case it is written by T2 and in the other case by T1.

$$X = 500$$

S_1	
T_1	T_2
$w_1(X, 100)$	
	$w_2(X, 200)$

$$X = 200$$

S_2	
T_1	T_2
	$w_2(X, 200)$
$w_1(X, 100)$	

$$X = 100$$

Conflicting Operations in a Schedule

- Notice that two read operations are not conflicting because changing their order makes no difference in outcome

$X = 100$

S_1	
T_1	T_2
$r_1(X)$	
	$r_2(X)$

S_2	
T_1	T_2
	$r_2(X)$
$r_1(X)$	

Serial Schedules

- A Simplified airline reservations database
- Each record includes the number of reserved seats among other information.
- a) shows a transaction T1 that transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y.
- b) shows a simpler transaction T2 that just reserves M seats on the first flight (X) referenced in transaction T1

(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);

Serial Schedules

- For example, two airline reservations agents submit to the DBMS transactions T1 and T2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:
  - 1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).
  - 2. Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).

Serial Schedules

- These two below schedules are called serial schedules

(a)

Time ↓

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

Time ↓

T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule B

Problems with Serial Schedules

- The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations.
- In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time.
- Additionally, if some transaction T is long, the other transactions must wait for T to complete all its operations before starting.
- Hence, serial schedules are unacceptable in practice.

Non Serial Schedules

- If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions.
- The concept of **serializability** of schedules is used **to identify which schedules are correct** when transaction executions have interleaving of their operations in the schedules.

(c)

Time ↓

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C

Time ↓

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D

Serial Schedules

- These two below schedules are called serial schedules

$X=90, Y=90$
$N=3, M=2$
$X=90-3=87$
$X=87$
$Y=90+3=93$
$X=87+2=89$
$X=89, Y=93$

(a)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

Time

T_1	T_2

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule B

$X=90, Y=90$
$N=3, M=2$
$X=90+2=92$
$X=92$
$X=92-3=89$
$Y=90+3=93$
$X=89, Y=93$

Non Serial Schedules

- The concept of **serializability** of schedules is used **to identify which schedules are correct** when transaction executions have interleaving of their operations in the schedules.

(c)

$X = 90, Y = 90$

$N = 3, M = 2$

$X = 90 - 3 = 87$

$X = 90 + 2 = 89$

$Y = 90 + 3 = 93$

$X = 92, Y = 93$

Time ↓

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C

Time ↓

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D

$X = 90, Y = 90$

$N = 3, M = 2$

$X = 90 - 3 = 87$

$X = 87$

$X = 87 + 2 = 89$

$Y = 90 + 3 = 93$

$X = 89, Y = 93$

Serializable Schedules

- Schedules that are always considered to be correct when concurrent transactions are executing.
- Such schedules are known as serializable schedules.
- A schedule S of n transactions is **serializable if it is equivalent to some serial schedule** of the same n transactions.
- Serializable schedules are always correct.

Serial and Non Serial Schedules

- Suppose we have i transactions T1, T2, … , Ti, and their number of operations are n_1, n_2, … , n_i, respectively.

$$Number\ of\ serial\ schedules = i\,!$$

$$Number\ of\ non\ serial\ schedules = \frac{(n_1+n_2+\ldots+n_i)\,!}{n_1\,! * n_2\,! * \ldots * n_i\,!}$$

Serializable Schedules

- Saying that a nonserial schedule S is serializable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct.
- The remaining question is: When are two schedules considered equivalent?
- There are several ways to define schedule equivalence.
  - Result equivalence
  - Conflict equivalence
  - View equivalence

Result Equivalent Schedules

- Two schedules are called result equivalent if they produce the same final state of the database.
- However, two different schedules may accidentally produce the same final state

$X = 100$
$X = 100 + 10 = 110$
$X = 110$

S_1
read_item(X);
$X := X + 10$;
write_item(X);

S_2
read_item(X);
$X := X * 1.1$;
write_item (X);

$X = 100$
$X = 100 * 1.1 = 110$
$X = 110$

$X = 200$
$X = 200 + 10 = 210$
$X = 210$

$X = 200$
$X = 200 * 1.1 = 220$
$X = 220$

Serializable Schedules

- The safest and most general approach to defining schedule equivalence is to focus only on the read_item and write_item operations of the transactions, and not make any assumptions about the other internal operations included in the transactions.
- For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order.
- Two definitions of equivalence of schedules are generally used: conflict equivalence and view equivalence.

Conflict Equivalence of Two Schedules

- If the **relative order of any two conflicting operations** is the same in both schedules then the two schedules are said to be **conflict equivalent**
- Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:
  - 1. they belong to different transactions
  - 2. they access the same item X
  - 3. at least one of the operations is a write_item(X)

Conflict Equivalence of Two Schedules

- For example, if a read and write operation occur in the order r1(X), w2(X) in schedule S1, and in the reverse order w2(X), r1(X) in schedule S2

$X=100$

S_1

T_1	T_2
$r_1(X)$	
	$w_2(X,200)$

$X=100$

S_2

T_1	T_2
	$w_2(X,200)$
$r_1(X)$	

$X=200$

read − write conflict

so S_1, S_2 are not conflict equivalent

- The value read by r1(X) can be different in the two schedules.

Conflict Equivalence of Two Schedules

- For example, if a write and write operation occur in the order w1(X), w2(X) in schedule S1, and in the reverse order w2(X), w1(X) in schedule S2

$X = 500$

S_1	
T_1	T_2
$w_1(X, 100)$	
	$w_2(X, 200)$

$X = 200$

S_2	
T_1	T_2
	$w_2(X, 200)$
$w_1(X, 100)$	

$X = 100$

$write - write\ conflict$

$so\ S_1, S_2\ are\ not\ conflict\ equivalent$

- The value of X can be different in the two schedules.

Serializable Schedules

- If two **conflicting operations** are applied in **different orders** in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are **not conflict equivalent**.
- Using the notion of conflict equivalence, we define a schedule S to be serializable if it is (conflict) equivalent to some serial schedule S′.
- In such a case, we can reorder the nonconflicting operations in S until we form the equivalent serial schedule S′.

Serializable Schedules

- Using the notion of conflict equivalence, we define a schedule S to be serializable if it is (conflict) equivalent to some serial schedule S′.

Schedule A ∧ Schedule D conflict Equivalent .

Schedule D is Conflict Equivalent t . o Serial schedule A

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

D is the Serializable schedule

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D

.

Serializable Schedules

- Using the notion of conflict equivalence, we define a schedule S to be serializable if it is (conflict) equivalent to some serial schedule S′.

C is not Serializable schedule

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C

T_1	T_2

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule B

Testing for Serializability of a Schedule using precedence graph

- There is a simple algorithm for determining whether a particular schedule is (conflict) serializable or not.
- The algorithm looks at only the **read_item** and **write_item** operations in a schedule to construct a precedence graph (or serialization graph)
- Which is a directed graph G = (N, E)
- that consists of a set of nodes N = {T1, T2, ... , Tn }
- And a set of directed edges E = {e1, e2, ... , em }.

Testing for Serializability of a Schedule using precedence graph

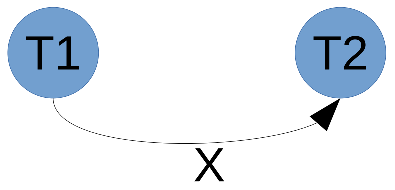
- Each edge ei in the graph is of the form ($T_j \rightarrow T_k$), $1 \leq j \leq n$,

  $1 \leq k \leq n$
- where Tj is the starting node of ei and Tk is the ending node of ei.
- Such an edge from node Tj to node Tk is created by the algorithm
  - if a pair of conflicting operations exist in Tj and Tk and the conflicting operation in Tj appears in the schedule beforethe conflicting operation in Tk.
- If there is a **cycle** in the precedence graph, schedule S is **not (conflict) serializable**;
- if there is **no cycle,** S is **serializable**

Testing for Serializability of a Schedule using precedence graph

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

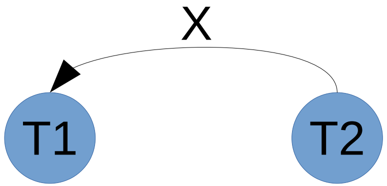


- if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable

Testing for Serializability of a Schedule using precedence graph

T_1	T_2
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule B

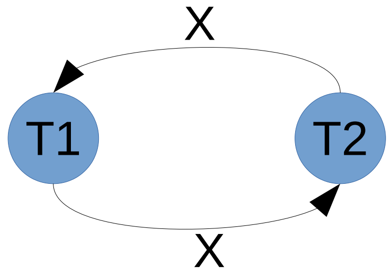


- if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable

Testing for Serializability of a Schedule using precedence graph

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C



Cycle → Not Coflict Serializable

- if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable

Testing for Serializability of a Schedule using precedence graph

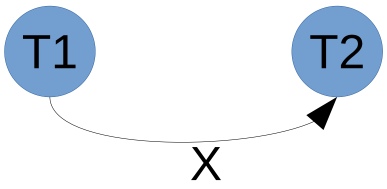
T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D



- if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable

View Equivalence and View Serializability

- Two schedules S and S′ are said to be view equivalent if the following three conditions hold:
  - 1. If the transaction T i in S reads an initial value for object X, so does the transaction T i in S’.
  - 2. If the transaction T i in S reads the value written by transaction T j in S for object X, so does the transaction T i in S’.
  - 3. If the transaction T i in S is the final transaction to write the value for an object X, so is the transaction T i in S’.

View Equivalence and View Serializability

- The idea behind view equivalence is that 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 read operations are hence said to see the same view in both schedules.
- Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules.
- A schedule S is said to be view serializable if it is view equivalent to a serial schedule.

View Equivalence and View Serializability

- conflict serializability and view serializability are similar if constrained write assumption(no **blind writes)** holds on all transactions in the schedule.
- This condition states that any write operation wi(X) in Ti is preceded by a ri(X) in Ti and that the value written by wi(X) in Tidepends only on the value of X read by ri(X).
- This assumes that computation of the new value of X is a function f(X) based on the old value of X read from the database.
- A **blind write** is a write operation in a transaction T on an item X that is **not dependent on the old value of X**, so it is **not preceded by a read of X** in the transaction T.

View Equivalence and View Serializability

- View serializability is less restrictive than that of conflict serializability under the **unconstrained write assumption(blind Write)**
- In below schedule w2(X) and w3(X) are blind writes, since T2 and T3 do not read the value of X.

$$S: \; r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$$

S

T_1	T_2	T_3
$r_1(X)$		
	$w_2(X)$	
$w_1(X)$		
		$w_3(x)$
c_1		
	c_2	
		c_3

View Equivalence and View Serializability

$$S: \; r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$$

S

T_1	T_2	T_3
$r_1(X)$		
	$w_2(X)$	
$w_1(X)$		
		$w_3(x)$
c_1		
	c_2	
		c_3

S'

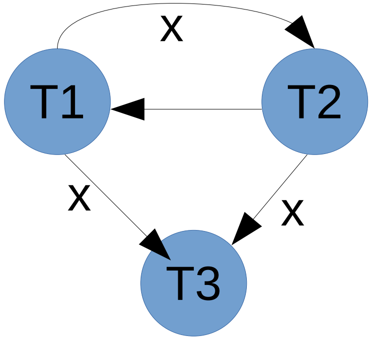
T_1	T_2	T_3
$r_1(X)$		
$w_1(X)$		
c_1		
	$w_2(X)$	
	c_2	
		$w_3(x)$
		c_3

The schedule S is view serializable, since it is view equivalent to the serial schedule S`(T1, T2, T3).

View Equivalence and View Serializability

$$S: \; r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$$

- The schedule S is view serializable, but not conflict serializable.
- It has been shown that any conflict-serializable schedule is also view serializable but not vice versa.
- the problem of testing for view serializability has been shown to be NP-hard



S		
T_1	T_2	T_3
$r_1(X)$		
	$w_2(X)$	
$w_1(X)$		
		$w_3(x)$
c_1		
	c_2	
		c_3

Characterizing Schedules Based on Recoverability

- For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved.
- In some cases, **it is even not possible to recover correctly after a failure**
- First, we would like to ensure that, once a transaction T is committed, it should never be necessary to roll back T
- This ensures that the durability property of transactions is not violated
- The schedules that theoretically meet this criterion are called **recoverable schedules**.

Characterizing Schedules Based on Recoverability

- A schedule where a committed transaction may have to be rolled back during recovery is called **non recoverable** and hence should not be permitted by the DBMS
- The condition for a recoverable schedule is as follows:
- A schedule S is recoverable if no transaction T in S commits until all transactions T′ that have written some item X that T reads have committed. (T reads some item X which is first written by T′)
- In a recoverable schedule, no committed transaction ever needs to be rolled back
- The durablity property of committed transaction should not be violated.

Characterizing Schedules Based on Recoverability

- The (partial) schedules S_a is recoverable.

$$S_a: \; r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

- because neither T1 nor T2 reads from other transaction, and none of the transactions committed.

S_a

T_1	T_2
$r_1(X)$	
	$r_2(X)$
$w_1(X)$	
$r_1(Y)$	
	$w_2(X)$
$w_1(Y)$	

Characterizing Schedules Based on Recoverability

- The (partial) schedule S_b is recoverable.

$$S_b: \; r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

- because T2 reads item X from T1, but T2 not committed yet.

S_a

T_1	T_2
$r_1(X)$	
$w_1(X)$	
	$r_2(X)$
	$w_2(X)$
$r_1(Y)$	
a_1	

Characterizing Schedules Based on Recoverability

- Consider the schedule S_a' given below, which is the same as schedule S_a except that two commit operations have been added to S_a:

$$S_a': \; r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$$

- S_a' is recoverable, even though it suffers from the lost update problem
- this problem is handled by serializability theory

S_a'

T_1	T_2
$r_1(X)$	
	$r_2(X)$
$w_1(X)$	
$r_1(Y)$	
	$w_2(X)$
	c_2
$w_1(Y)$	
c_1	

Characterizing Schedules Based on Recoverability

- The schedule S_c is not recoverable.
- S_c is not recoverable because T2 reads item X from T1, but T2 commits before T1commits.
- The problem occurs if T1 aborts after the c2 operation in S_c;
- then the value of X that T2 read is no longer valid and T2 must be aborted after it is committed, leading to a schedule that is not recoverable.

$$S_c: \; r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$$

S_c

T_1	T_2
$r_1(X)$	
$w_1(X)$	
	$r_2(X)$
$r_1(Y)$	
	$w_2(X)$
	c_2
a_1	

Characterizing Schedules Based on Recoverability

- The schedule S'$_c$ is recoverable.

$$S_c^{'}:\ r_1(X);w_1(X);r_2(X);r_1(Y);w_2(X);a_1;a_2;$$

- If T1 aborts instead of committing, then T2 should also aborted, because the value of X it read is no longer valid.
- In S'$_c$, aborting T2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule S$_c$

$S_c^{'}$

T_1	T_2
$r_1(X)$	
$w_1(X)$	
	$r_2(X)$
$r_1(Y)$	
	$w_2(X)$
a_1	
	a_2

Characterizing Schedules Based on Recoverability

- The schedule S'c is recoverable.

$$S_c^{'}:\ r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_1; c_2;$$

- Sc is recoverable because T2 reads item X from T1, and T2 commits after T1 commits.

$S_c^{'}$	
T_1	T_2
$r_1(X)$	
$w_1(X)$	
	$r_2(X)$
$r_1(Y)$	
	$w_2(X)$
c_1	
	c_2

Cascading Rollback

- In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated.
- However, it is possible for a phenomenon known as cascading rollback (or cascading abort) to occur in some recoverable schedules, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.
- This is illustrated in schedule S'_c, where transaction T2 has to be rolled back because it read item X from T1, and T1 then aborted.

Cascading Rollback

- The schedule S’$_c$ is recoverable.

$$S_c': \; r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); a_1; a_2;$$

- Cascading rollback
- Because cascading rollback can be time consuming since numerous transactions can be rolled back
- It is important to characterize the schedules where this phenomenon is guaranteed not to occur.

S_c'

T_1	T_2
$r_1(X)$	
$w_1(X)$	
	$r_2(X)$
$r_1(Y)$	
	$w_2(X)$
a_1	
	a_2

Cascadeless Schedule

- A schedule is said to be cascadeless, or to avoid cascading rollback, if every transaction in the schedule reads only items that were written by committed transactions.
- In this case, all items read will not be discarded because the transactions that wrote them have committed, so no cascading rollback will occur.

Cascadeless Schedule

- The r2(X) command in schedules Sc must be postponed until after T1 has committed (or aborted)
- Thus delaying T2 but ensuring no cascading rollback if T1 aborts.

S_d	
T_1	T_2
$r_1(X)$	
$w_1(X)$	
$r_1(Y)$	
c_1/a_1	
	$r_2(X)$
	$w_2(X)$

Strict Schedule

- Finally, there is a third, more restrictive type of schedule, called a **strict schedule**, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).
- Strict schedules simplify the recovery process. In a strict schedule, the process of undoing a write_item(X) operation of an aborted transaction is simply to restore the before image (old_value or BFIM) of data item X.
- This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules.

Strict Schedule

$$S_f: \ w_1(X,5); w_2(X,8); a_1;$$

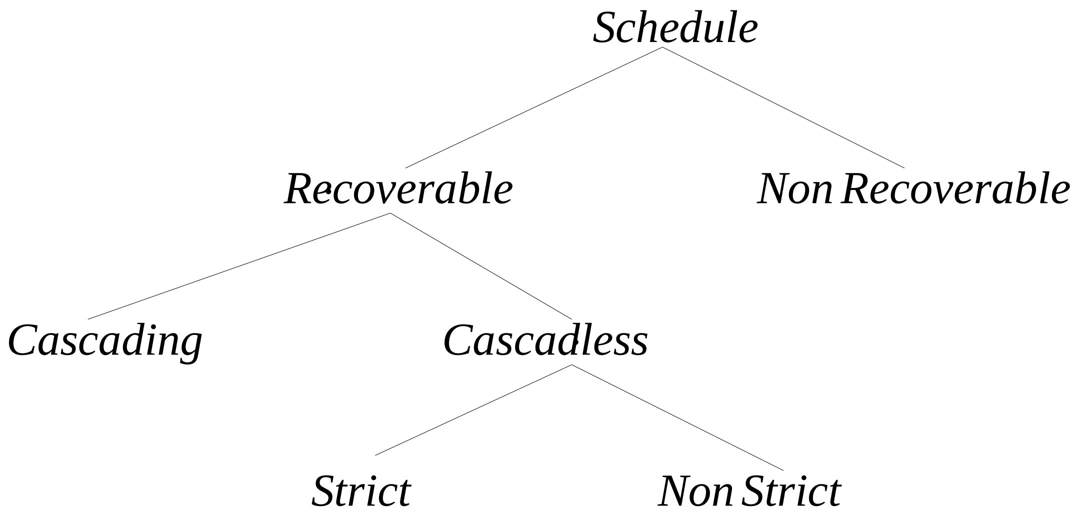
S_e	
T_1	T_2
$w_1(X,5)$	
	$w_2(X,8)$
a_1	

- Consider schedule Sf:
- Suppose that the value of X was originally 9, which is the before image stored in the system log along with the w1(X, 5) operation.
- If T1 aborts, as in Sf, the recovery procedure that restores the before image of an aborted write operation will restore the value of X to 9, even though it has already been changed to 8 by transaction T2, thus leading to potentially incorrect results.
- Although schedule Sf is cascadeless, it is not a strict schedule, since it permits T2 to write item X even though the transaction T1 that last wrote X had not yet committed (or aborted). A strict schedule does not have this problem.

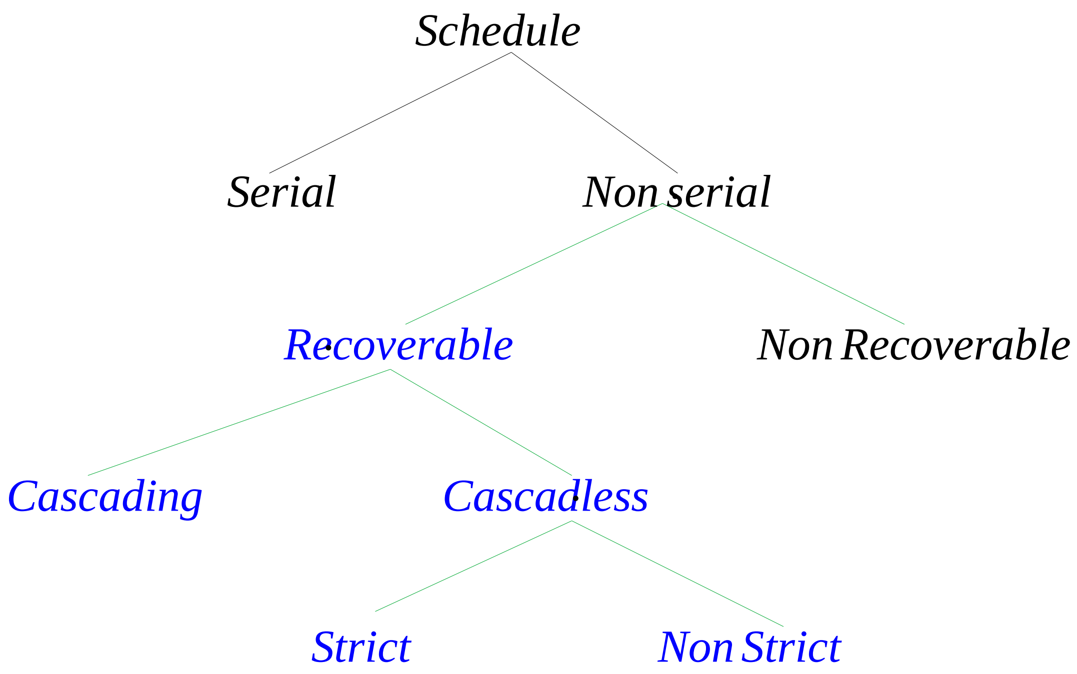
Characterizing Schedules Based on Recoverability

- It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable.
- Suppose we have i transactions T1, T2, ... , Ti, and their number of operations are n1, n2, ... , ni, respectively.
- If we make a set of all possible schedules of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable.
- The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules.
- Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.
- Most recovery protocols allow only strict schedules, so that the recovery process itself is not complicated.

Characterizing Schedules Based on Recoverability



Classification of Schedules



Exercise Problems

- Consider the following classes of schedules:***serializable, conflict-serializable,view-serializable, recoverable, avoids-cascading-aborts,andstrict***.
- For each of the followingschedules, state which of the above classes it belongs to.

1. T1:R(X), T2:R(X), T1:W(X), T2:W(X)
2. T1:W(X), T2:R(Y), T1:R(Y), T2:R(X)
3. T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y)
4. T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y)

Exercise Problems

5. T1:R(X), T2:W(X), T1:W(X), T2:Abort, T1:Commit
6. T1:R(X), T2:W(X), T1:W(X), T2:Commit, T1:Commit
7. T1:W(X), T2:R(X), T1:W(X), T2:Abort, T1:Commit
8. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Commit
9. T1:W(X), T2:R(X), T1:W(X), T2:Commit, T1:Abort
10. T2: R(X), T3:W(X), T3:Commit, T1:W(Y), T1:Commit, T2:R(Y), \ T2:W(Z), T2:Commit
11. T1:R(X), T2:W(X), T2:Commit, T1:W(X), T1:Commit, T3:R(X), T3:Commit
12. T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:Commit, T2:Commit, T3:Commit