Schedule

A sequence of operations (read, write, commit, rollback) by a set of concurrent transactions that preserves the order of operations in each of the individual transactions

(T1, R(x)), (T1, W(x))
(T2, R(y)), (T2, W(y))

(T1, R(x)), (T2, R(y)), (T2, W(y)), (T1, W(x))
(T1, R(x)), (T2, R(y)), (T1, W(x)), (T2, W(y))

Serial Schedule

A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions

(T1, R(x)), (T1, W(x))
(T2, R(y)), (T2, W(y))

(T1, R(x)), (T1, W(x)), (T2, R(y)), (T2, W(y)) // T1, T2
(T2, R(y)), (T2, W(y)), (T1, R(x)), (T1, W(x)) // T2, T1

Nonserial Schedule

(T1, R(x)), (T1, W(x))
(T2, R(y)), (T2, W(y))

// neither T1,T2 nor T2, T1
(T1, R(x)), (T2, R(y)), (T2, W(y)), (T1, W(x))
(T1, R(x)), (T2, R(y)), (T1, W(x)), (T2, W(y))
(T2, R(y)), (T1, R(x)), (T1, W(x)), (T2, W(y))
(T2, R(y)), (T1, R(x)), (T2, W(y)), (T1, W(x))

Serializability

- Two serial schedules need not produce the same result.
- However, a serial schedule will always leave the database in a consistent state.
- Therefore, every serial execution is considered correct.
- The objective of serializability is to find nonserial schedules to execute concurrently without interfering one other, and thereby produce a database state that could be produced by a serial schedule.

Serializability

We call a schedule **serializable** if it has the same effect as some serial schedule regardless of the specific information in the database.

Definition of serializability is a bit difficult to handle: How can we test for the same effect regardless of data?

To come up with an answer, we'll create a stricter definition of serializability, called *conflict-serializability*.

Conflict

In serializability the ordering of read and write operations are important.

If two transactions only read a data item there is no conflict. (Ti , R(x)), (Tj, R(x))

If two transactions read/write separate data item there is no conflict. (Ti , R/W(x)), (Tj, R/W(y))

If two transactions read/write the same data and if one of the operations is write, there is a conflict

(Ti , R(x)), (Tj, W(x)) // conflict

(Ti , W(x)), (Tj, R(x)) // conflict

(Ti , W(x)), (Tj, W(x)) // conflict

Conflict-equivalence

Two schedules are **conflict-equivalent** if one can be reached from the other through a series of swaps of adjacent operations, where no swap falls into one of the following patterns:

- the operations are by the same transaction (why?)
- the operations use the same database element, and at least one is a write (why?)

Conflict-equivalence

Example

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

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

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

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

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

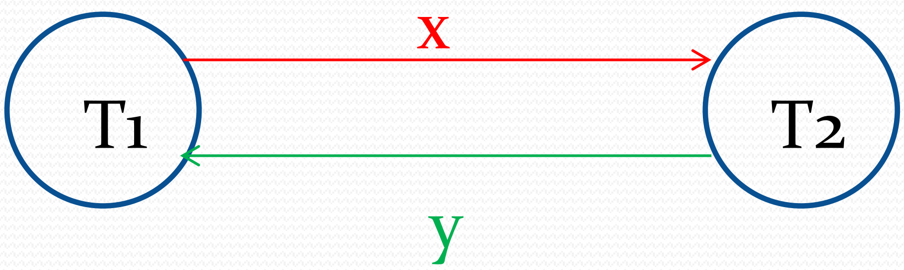
T2, T1

Conflict Serializability

A schedule is **conflict-serializable** if it is conflict-equivalent to some serial schedule.

Example: Nonconflict serializable schedule

(T1, R(x)), (T1, W(x)), (T2, R(x)), (T2, W(x)), (T2, R(y)), (T2, W(y)), (T1, R(y)), (T1, W(y))



Precedence graph has a cycle

Example: Serializable

The nonserial schedule S (Assume x = a, y = b)

(T1, R(X)), (T2, R(Y)), (T3, W(X)), (T2, R(X)),

a b c c

(T1, R(Y)), Commit(T1), Commit(T3), Commit(T2)

b

This is equivalent to T1,T3,T2

(T1, R(X)), (T1, R(Y)), Commit(T1), (T3, W(X)),

a b c

Commit(T3), (T2, R(Y)), (T2, R(X)), Commit(T3)

b c

Thus, S is serializable.

Example: Conflict Serializable

Condider the nonserial schedule S

(T_1, R(X)), (T_2, R(Y)), (T_3, W(X)), (T_2, R(X)), (T_1, R(Y))

S is conflict-equivalent to

(T_1, R(X)), (T_1, R(Y)), (T_3, W(X)), (T_2, R(Y)), (T_2, R(X))

Thus S is conflict-equivalent to the serial schedule T_1,T_3,T_2.

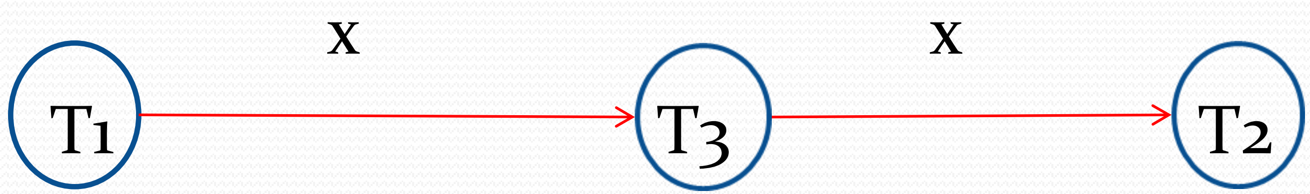
So, S is conflict serializable.

Example: Conflict Serializable

Condider the nonserial schedule S

(T1, R(X)), (T2, R(Y)), (T3, W(X)), (T2, R(X)), (T1, R(Y))

An alternate way to determine whether or not S is conflict serializable is to create a precedence graph.



Since there is no cycle, S is conflict serializable. Futher, note that precedence graph gives the conflict equivalent serial schedule.

Example: Serializable but not conflict serializable

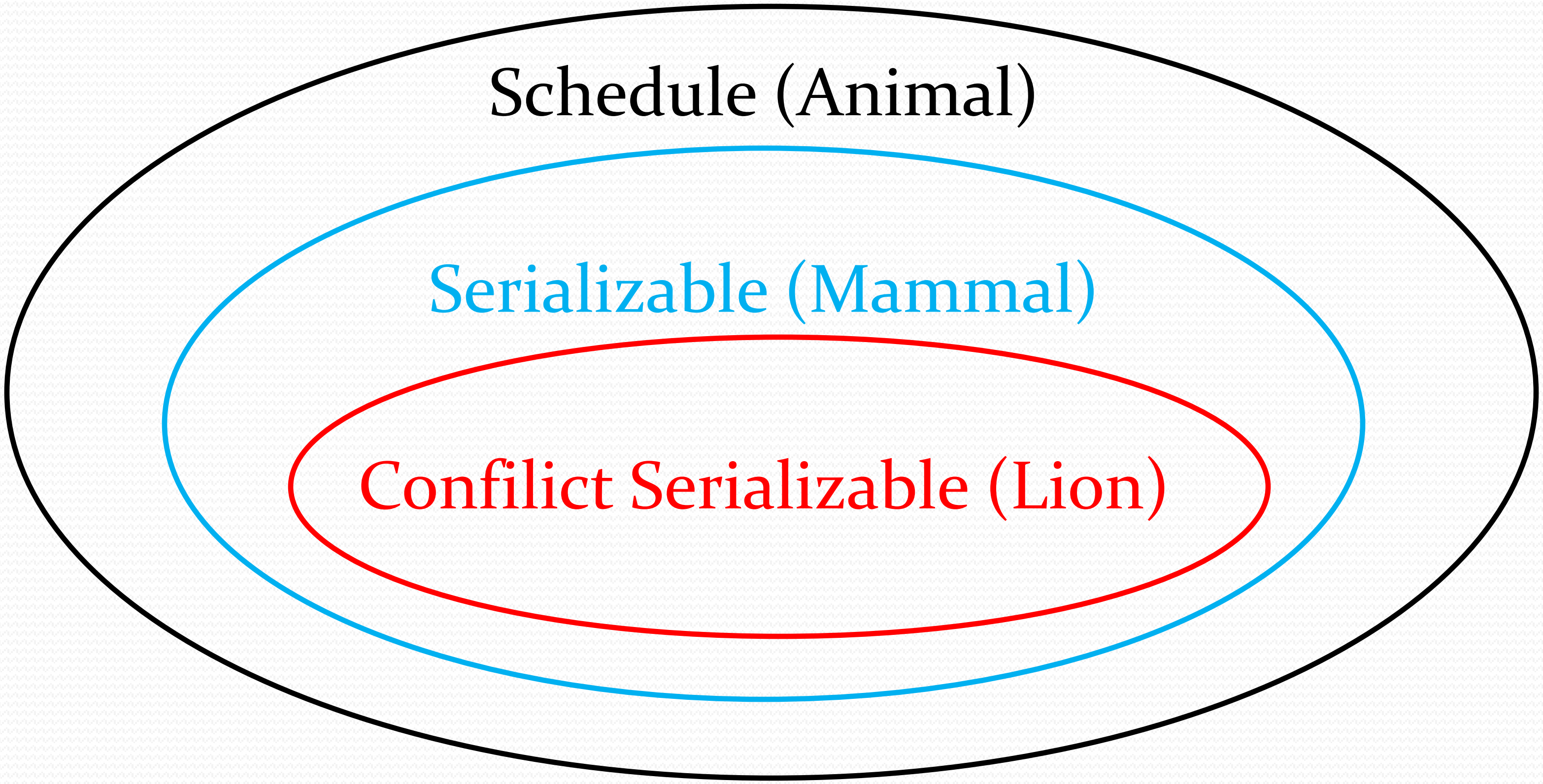
(T1, W(*A*)),(T2, W(*A*)), (T2, W(*B*)),(T1, W(*B*)),(T3, W(*B*))

No swap is possible, so there is no serial schedule that is conflict-equivalent to this one. (Also note the precedence graph has a cycle T1->T2->T1. So it is not conflict serializable or nonconflict serializable.)

However, since no transaction ever reads the values written by the (T1, W(A)),(T2, W(B)) and (T1, W(*B*) operations, the schedule has the same outcome as the serial schedule, T1, T2, T3.

(T1, W(*A*)),(T1, W(*B*)),(T2, W(*A*)), (T2, W(*B*)),(T3, W(*B*))

Schedule, Serializable, Conflict Serializable



2PL

There are two types of locks:

Read_lock (or shared lock) and Write_lock (or exclusive lock)

Two possible scenarios:

ZERO Read_lock, ONE Write_lock

MANY Read_lock, ZERO Write_lock

- A transaction must have the Read_lock to Read
- A transaction must have the Write_lock to Write. If a transaction has a Write_lock, it can read as well.
- A transaction must WAIT, if the necessary lock is not available.
- All lock are held until commit and released after commit.

Example: 2PL

(t1, START (T1))	
(t2, read(T1, X))	// T1 has Read lock on X
(t3, START (T2))	
(t4, read(T2, Y))	// T2 has Read lock on Y
(t5, START (T3))	
(t6, write(T3, X))	// T3 WAIT for Write lock on X
(t7, read(T2, X))	// T2 has Read lock on X
(t8, read(T1, Y))	// T1 has Read lock on Y
(t9, commit(T1))	// All locks held by T1 are released
(t10, commit(T2))	// All locks held by T2 are released
(t11, write(T3, X))	// T3 gets write lock on X
(t12, commit(T3))	// All locks held by T3 are released

(Single Version) Timestamping

Read(X)

If X is **written** by an younger transaction

Abort, Rollback, Restart

Else

Read //read_ts = max{ts(T), read_ts}

Write(X)

If X is **read** by an younger transaction

Abort, Rollback, Restart

Else If X is written by an younger transaction

Ignore Write

Else

Write // set write ts as ts(T)

(Single Version) Timestamping

The nonserial schedule S (Variable(readTS, writeTS))

(t_1, START (T_1)),

(t_2, read(T_1, X)), X(t_1,_)

(t_3, START (T_2)),

(t_4, read(T_2, Y)), Y(t_3, _)

(t_5, START (T_3)),

(t_6, write(T_3, X)), X(t_1, t_5)

(t_7, read(T_2, X)), t_3 = TS(T_2) < writeTS(X) = t_5, Abort, Roll back, Restart

(t_8, read(T_1, Y)), Y(t_3, _) t_3 = max{t_3, t_1}

(t_9, commit(T_1))

(t_{10}, START(T_2))

(t_{11}, commit(T_3))

(t_{12}, read(T_2, Y)), Y(t_{10}, _)

(t_{13}, read(T_2, X)), t_{10} = TS(T_2) >= writeTS(X) = t_5, X(t_{10}, t_5)

(t_{14}, commit(T_2)) // t_{10} = max{t_1, t_{10}}

(Multi-Version) Timestamping

Read(X)

Pick the version.

The version with largest write_ts <= ts(T)

Read // Read never fails

Write(X)

Pick the version.

The version with largest write_ts <= ts(T)

If read_ts <= TS(t) //not read by younger trans.

Create new version

//set read and write ts ts(T)

Else

Abort, Rollback, Restart

(Multi-Version) Timestamping

The nonserial schedule S (VARversion(readTS, writeTS))

(t1, START (T1)), // Assume Xo(to, to), Yo(to, to)

(t2, read(T1, X)), Xo(t1, to)

(t3, START (T2)),

(t4, read(T2, Y)), Yo(t3, to)

(t5, START (T3)),

(t6, write(T3, X)), X1(t5, t5)

(t7, read(T2, X)), // Xo selected Xo(t3, to)

(t8, read(T1, Y)), Yo(t3, to) t3 = max{t3, t1}

(t9, commit(T1))

(t10, commit(T2))

(t11, commit(T3))

Checkpoint

	Deferred Update	Immediate Update
Transaction was committed before checkpoint	No Rollback No Roll forward No Restart	No Rollback No Roll forward No Restart
Transaction was committed after checkpoint	Roll forward	Roll forward
Transaction never committed	No Rollback No Roll forward Restart	Rollback Restart