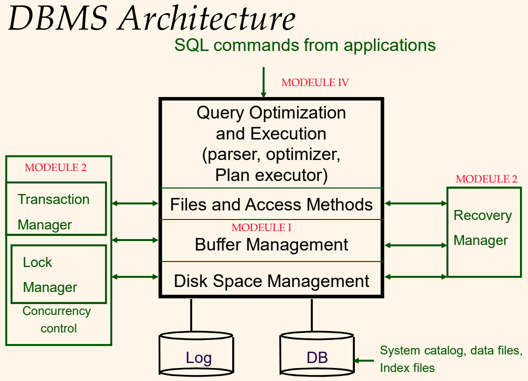
Transaction Management

Chapters 16 and 17

Instructor: Sharma Chakravarthy
sharma@cse.uta.edu
The University of Texas @ Arlington

DBMS Architecture



What is a Transaction or Tx?

- A transaction is a program or application written in some programming language that includes reading and modifying a database.
  - An SQL query
  - An SQL Insert/delete/update
  - Applications with embedded SQL
  - Stored procedures
  - User defined functions
  - …
- No restriction on the size of a Tx
- No synchronization primitives are used for writing applications!

Concurrency control and Recovery

- Concurrency Control
  - the activity of coordinating the actions of processes that operate concurrently (at the same time), access shared data, and therefore potentially interfere with each other
- Recovery
  - the activity of ensuring that software and hardware failure do not corrupt persistent data
  - the database contains all the effects of **committed** transactions and none of the effects of **uncommitted** transactions
- Parallel and concurrent executions are not the same
  - Parallel implies no switch (typically on different processors)
  - Concurrent means sharing a processor and switching tasks
- DBMSs also do parallel processing using multiple processors

Relationship with the mutual exclusion

- Mutual exclusion is from OS
- OS does concurrent scheduling of tasks and manage multiple users!
- In multi-threaded applications, synchronization need to be managed by the person writing the code.
  - In a DBMS, applications are not even aware of it.
  - It is managed by the DBMS
- MyMav registration is a good example

ACID Properties of transaction

- Atomicity: all or nothing property
  - Who is responsible for this?
- Consistency: consistent DB state + correct transactions ➔ consistent DB state!!
  - Who is responsible for this?
- Isolation: Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.
  - Who is responsible for this?
- Durability: Once a transaction completes successfully (or **commits)**, its changes to the DB survive failures.
  - Who is responsible for this?

Requirements of TM Systems

- High Performance – measured in transactions per second (TPS); dollars per transactions
- High Availability – ability to provide access to users all the time (24 x 7 operation)
- Correctness – ability to provide correct results even in the face of failures (of any kind)
- Support various levels (degrees) of consistency
- For replicated databases, mutual consistency should also be maintained

Why Concurrency Control ?

- Concurrency control can
  - increase processor utilization (why?)
  - increase total transaction throughput (why?)
  - may increase response time (slightly) for individual transactions! Why?
  - Short transactions do not get delayed due to long running transactions Why?
- The above is especially important in a DBMS where transactions access data from secondary storage devices (CPU is waiting for a disk read to complete!!)
  - Remember impedance mismatch!

Why Recovery ?

- Needed to accommodate various kinds of failures
  - logical errors (abort by the transaction/application)
  - system errors (abort by the system – due to deadlock)
  - system crashes: losing the contents of volatile storage
  - Power failures
  - loss of non-volatile storage (or media failure) ??
  - Others (disasters)
    - Mirroring, hot standby!

Terminology

- Both concurrency control and recovery are applied to transactions - an arbitrary collection of database operations (read/write operations) specified by an application.
- A transaction is an execution of a program that accesses a shared database

The goal of Concurrency control and Recovery

- is to ensure that transactions execute atomically, meaning that:
  - each transaction accesses shared data without interfering with other transactions (isolation), and
  - if a transaction terminates normally, then all its effects are made permanent; otherwise it has no effect at all.

Operations

- The ACID properties are usually ensured by combining two different sets of algorithms
  - Concurrency control protocols
    - Ensure isolation property, and
    - Consistency of Tx execution (based on atomicity)
  - Recovery protocols
    - Ensure atomicity, and
    - Durability
- What about consistency of a Tx?
  - Who is responsible for the consistency of a Tx?

Operations

- Read, Write, Commit, Abort
  - Each transaction is assumed to be self contained; i.e., there is no direct communication with other transactions. However, transactions do communicate indirectly by manipulating shared data in the database.
  - Executing a transaction's commit constitutes a guarantee by the DBMS that it will not abort that transaction and that the transaction's effects will survive subsequent failures of the system

Operations contd.

- When a transaction T aborts (by its own choice or done by the system), the system must wipe out all of its effects; there are two kinds of effects:
  - i) On data: That is, values that T wrote in the database and
  - ii) On other transactions. That is, transactions (if any) that read values written by T.
  - Both (i) and (ii) should be dealt with
  - ii) may, in turn, cause other transactions to be aborted leading to a phenomenon termed cascading aborts.

Concurrency Problems

- Arise due to interleaved execution of transactions.
- If transactions are executed sequentially one after another (i.e., serially), then there is no problem as each transaction is assumed to preserve the consistency of the database.
- Clearly understand the difference between
  - Serial/sequential executions and
  - Serialized or serializable (not serial, but behaves like serial) execution!

Example:

Assume a = 1000; b=1000

Correctness: a+b should be 2000 at the end of execution of execution of T1 and T2 (T1 followed by T2 or T2 followed by T1 or interleaved!)

Consider T1 and T2.

```
T1:  read(a)                T2:  read(a)
     a:=a-50                     temp = a*0.1
     write(a)                    a = a-temp
     read(b)                     write (a)
     b:= b+50                    read (b)
     write(b)                    b = b+temp
                                 write (b)
```

Transfers 50 from account a to account b

Transfers 1% from account a to account b

Shedule 1: T1 followed by T2

read(a)
a:=a-50
write(A)
read(b)
b:= b+50
write(b)

read(a)
temp = a*0.1
a = a-temp
write (a)
read (b)
b = b+temp
write (b)

a = 855

b = 1145

Is this correct?
If so why?

This is a serial execution (schedule) !!

Is this a serializable execution (schedule) ?

Shedule 2: T2 followed by T1

read(a)
temp = a*0.1
a = a-temp
write (a)
read (b)
b = b+temp
write (b)

read(a)
a:=a-50
write(A)
read(b)
b:= b+50
write(b)

a = 850

b = 1150

a+ b is 2000

Is this Correct?
If so why?

This is a serial schedule !!

Is this a serializable schedule ?

Schedule 3: an interleaved schedule of T1 and T2

T1	T2
read(a)	
a:=a-50	
write(A)	
	read(a)
	temp = a*0.1
	a = a-temp
	write (a)
read(b)	
b:= b+50	
write(b)	
	read (b)
	b = b+temp
	write (b)
a = 855	b = 1145

Is this correct?
If so why?

Is this is a serializable schedule ?

Is every serializable schedule equivalent to a serial schedule?

Shedule 4: another interleaved schedule of T1 and T2

read(a)	
a:=a-50	
	read(a)
	temp = a*0.1
	a = a-temp
	write (a)
	read(b)
write(a)	
read(b)	
b:= b+50	
write(b)	
	b = b+temp
	write (b)
	b = 2100
a = 950	

Is this correct?
If so why?

Is this is a serializable schedule ?

Why are we interested in serializable and non-serializable schedules? Instead of serial schedules!

Concurrency Problems

- The inconsistency of the intermediate states is the source of concurrency control problems

R
W → dependencies
A
C

W → W
W → R → dependencies
R → W

R → R

A and C will play a role in recovery, also need to be taken into account for concurrency!

Concurrency Problems

- Lost update problem
  - If transaction T2 updates a record previously updated by a transaction T1 then undoing T1 will also undo the update of T2. This is called write → write dependency
  - Even if T1 commits, T2 would have read an uncommitted value!!

There may or may not be a read update(y) abort

T1 x

There may or may not be a read update(y) commit

T2 x x

time

Concurrency Problems contd.

- Dirty read/temporary update problem
  - Reading uncommitted data
- Write → Read dependency

T1	T2
Read(x)	
x=x-N	
write(x)	
	Read(x)
	x=x+M
	write(x)
Read/write(y)	

time

Transactions can create inconsistent state before they commit. Even if both transactions commit, T2 has read an inconsistent value and acted upon it.

Concurrency Problems contd.

- **Unrepeatable read problem**
- Read → Write Dependency
-

T1	T2
Read(x)	
Print(x)	
	Read(x)
	x=x+M
	Write(x)
Read(x)	.
Print(x)	.

T1 reads a value of x different from the value of x read earlier (although x has not been changed by T1)

Concurrency Problems contd.

- In the absence of concurrent execution, none of the above anomalies would arise.
- Note: there is no problem with Read →Read dependency as reads do not change value. They commute; i.e., order of reads do not change the database state
- Hence, many readers can allowed in a critical section, but only one writer!

Jim Gray's Laws

- First Law of concurrency Control
  - Concurrent execution should not cause application programs to malfunction
- Second law of concurrency Control
  - Concurrent execution should not result in lower throughput or much higher response time than serial execution

Use simple algorithms !!

Consistency levels

- Transaction *T sees* ***degree 3*** *consistency* (serializability) if
  - 1. *T* does not overwrite dirty data of other transactions
    (avoids w-w dependency)
  - 2. *T* does not read dirty data from other transactions
    (avoids w-r dependency)
  - 3. Other transactions do not dirty any data read by *T* before *T* completes (avoids r-w dependency)
  - *4. T* does not make visible (commit) any writes until it completes all its writes (i.e., until the end of transaction (EOT))
- Transaction *T sees* ***degree 2*** *consistency (cursor stability)* if
  - 1. *T* does not overwrite dirty data of other transactions (w-w)
  - 2. *T* does not read dirty data from other transactions (w-r)
  - 4. *T* does not commit any writes before EOT.

Consistency levels contd.

- Transaction *T sees* ***degree 1*** *consistency* (browse) if:
  - 1. *T* does not overwrite dirty data of other transactions. (w-w)
  - 4. *T* does not commit any writes before EOT.
- Transaction *T sees* ***degree 0*** *consistency (anarchy)* if:
  - 1. *T* does not overwrite dirty data of other transactions (w-w)

  i.e., you wait for other transactions to commit ONLY for writing. Dirty reads ok. Unrepeatable reads also ok.

Of course, a higher degree of Consistency encompasses all the lower degrees

A Comment on Degree of Consistency

- All DBMSs support all the 4 degrees and you can indicate degree with each Tx
- *It is felt that supporting degree 3 which is needed for some applications (e.g., banking, airline reservation, payroll) is not needed for all applications*
- *NoSQL DBMSs trade of ACID properties with CAP (consistency, availability, and partitioning) functionality*
- *Eventual consistency (different from mutual consistency) is supported for partitions!*
- *Recovery may also be done in a less stringent manner! As fault tolerance!*

Serializability

- One way to avoid interference is to NOT allow transactions to be interleaved at all. An execution in which no two transactions are interleaved is called **serial** or **sequential**
- More precisely, an execution is called serial, if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other

Serializability contd.

- A serial execution provides atomic (all or nothing) processing of transactions (assuming recovery)
- Serial executions are correct by definition because each transaction individually is correct (by assumption) and transactions that execute serially cannot interfere with each other.
- Not all serial executions produce the same effect on the database state
  - e.g. (i) T1 followed by T2 and (ii) T2 followed by T1 (see earlier example)
  - T1, T2 gave a=855, b = 1145; T2, T1 gave a=850, b=1150!

  Although (i) and (ii) produce different database states, both are equally correct and acceptable

  - This is different from expert systems/AI where conflict resolution strategies are used to choose the order of evaluation!

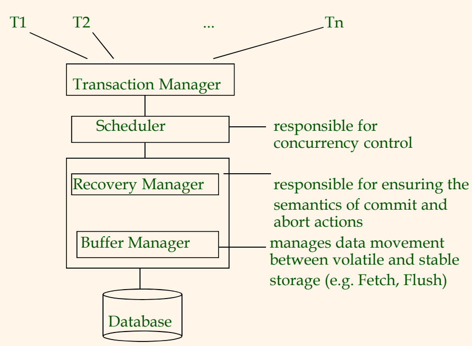
Serializability contd.

- The class of allowable executions can be broadened to include executions that have the same effect as serial ones. Such executions are called ***Serializable***.
- More precisely, an execution of a set of transactions is *serializable* if it produces the same output and has the same effect on the database as some serial execution of the same set of transactions.

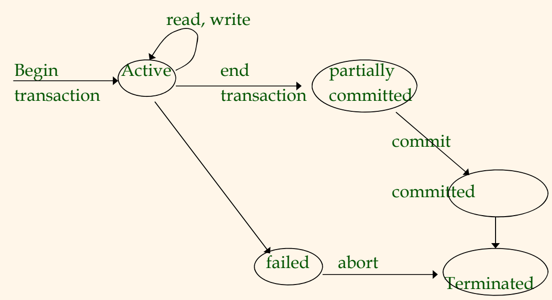
Serializability contd.

- Since all serial executions are correct, and since each serializable execution has the same effect as a serial execution, serializable executions are correct too.
- All serializable executions are equally correct. Therefore, the DBMS may execute transactions in any order, as long as the effect is the same as that of some serial order.
- Serial execution is used as the ground truth!

ABSTRACT SYSTEM MODEL



States of a transaction



Serializability Theory

- It is a mathematical tool that allows us to prove whether or not a schedule works correctly. For this, we represent a concurrent execution of a set of transactions by a structure called History (also known as schedule/log/audit)
  - It is essentially a partial order of Tx's operations!

Schedule

- Definition: A schedule for a set of transactions {T1, T2, ..., Tn} is a sequence of actions constructed by merging the actions of T1, T2, ..., Tn while respecting the order of the actions making up each transaction

T1: a11, a12

T2: a21, a22

S1 = a11, a21, a22, a12 a correct schedule

(may or may not be a serializable schedule)

S2 = a12, a21, a22, a11 Not a correct schedule

We will not consider incorrect schedules!

Serial Schedule

- Serial Schedule: A schedule of a set of transactions {T1,T2,...,Tn} is a serial schedule if there exists a permutation π of {1,2,...,n} such that S = $<T_{\pi(1)}, T_{\pi(2)}, \ldots, T_{\pi(n)}>$
  - How many serial schedules are there for n transactions?
- Serializable schedule: A schedule of a set of transactions T1,T2, ..., Tn is serializable if it yields exactly the same results (database state) as a serial schedule of {T1, T2, ..., Tn}.
  - How many serializable schedules are there for n transactions?

Number of Serial and Interleaved Schedules

- Given n Txs {T1,T2,...,Tn}

1. How many serial schedules (SS) are there?

   n!

2. How many interleaved (legal or not) schedules are there? (assume k ops in each T)

   (nk)! >> n! (let us say IS)

3. How many correct/legal interleaved schedules are there?

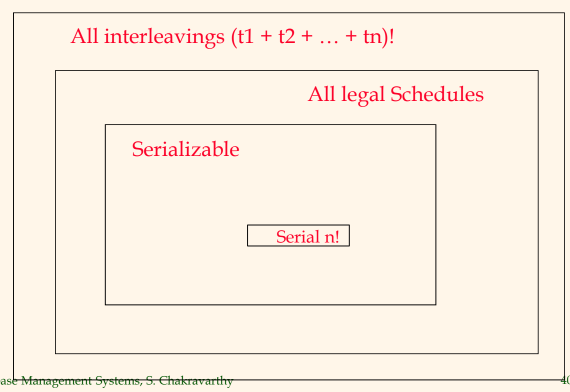
   between n! and IS (let us say CIS, closer to n!)

4. How many Serializable schedules are there?

between n! and CI (this is of interest to us)

<< CIS

Venn Diagram of serializability



Selializability flavors

- Serializability: only says that the execution should be equivalent to some serial execution. That is, gives the same result (actually state) as that of some serial schedule.
- Conflict serializability is based on the notion of conflicting operations! A read operation conflicts with write (on the same object) and a write conflicts with both read **and** write (on the **same object**).
- View seriallizability: based on what each Tx reads and writes (sees) in a schedule. Stronger than conflict serializability and weaker than serializability.

Selializability flavors

- Serializability: no easy way to check! (why?)
- Conflict serializability: can be checked using the notion of conflicting operations.
  - Two operations are said to conflict if they both operate on the same data item and at least one of them is a write. Thus

    Read (x) conflicts with Write(x), while

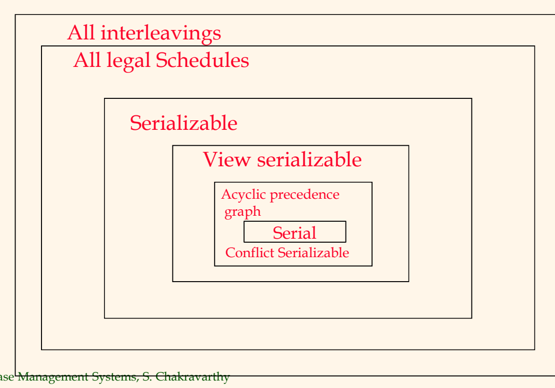
    Write(x) conflicts with both Read(x) and Write(x)

    Read does not conflict with another Read
- View serializability: there are well-defined conditions for checking for this.

Selializability flavors

- Serializability flavors are not the same.
  - Every conflict serializable schedule is serializable (important)
  - However, the converse is not true. (not so important)
  - Conflict serializability is sufficient but not necessary for serializability (see example)
  - Every conflict serializable schedule is also view serializable. However, the converse is not true (see example)
  - Also, every view serializable schedule is serializable. However, the converse is not true (see example)

Venn Diagram of serializability



Definition for serializability flavors

- A schedule is serializable if it is equivalent to a serial schedule (i.e., produces the same result)
- A schedule is conflict serializable if it is conflict equivalent to some serial schedule.
  - Conflict equivalent means that all pairs of conflicting operations appear in the same order in both schedules!
- A schedule is view serializable if it is view equivalent to a serial schedule. That is, satisfies 3 conditions of view serializability

Example 1 *(serializable without being conflict equivalent)*

- Consider the following schedules:

T1: R(A)			T1: R(A)
	T2: W(A)		W(A)
	Commit		commit
W(A)			T2: W(A)
Commit			commit
		T3: W(A)	T3: W(A)
		commit	commit

- The above is NOT conflict serializable as conflicting operations T2:W(A) and T1:W(A) are not in the same order!
- However, the above is equivalent to a serial schedule [(T1, T2, T3) or (T2, T1, T3)]
- Shows that serializability does not imply conflict serializability. However, the converse is true
- Blind writes creates this problem!

View Serializability

- A schedule is view serializable if it is view equivalent to some serial schedule
- Two schedules S1 and S2 on Ti and Tj are view equivalent:
  1. If Ti reads the initial value of object A in S1, it must also read the initial value of A in S2
  2. If Ti reads a value of A written by Tj in S1, it must also read the value of A written by Tj in S2
  3. For each data object A, the transaction (If any) that performs the final write on A in S1 must also perform the final write on A in S2
- Previous example is view serializable, but NOT conflict serializable. It can be shown that any view serializable schedule that is NOT conflict serializable contains a blind write.

Example 1 *(view serializable without being conflict serializable)*

- Consider the following schedules:

T1: R(A)			T1: R(A)
	T2: W(A)		W(A)
	Commit		commit
W(A)			T2: W(A)
Commit			commit
		T3: W(A)	T3: W(A)
		commit	commit

- The above is NOT conflict serializable as conflicting operations T2:W(A) and T1:W(A) are not in the same order!
- However, the above is view serializable as it satisfies all 3 conditions: 1) A is read by T1 in both; 3)A is the last item written in both; 3) no one reads a value written by others!
- Shows that view serializability does not imply conflict serializability.

Example 2

- T1: R(A)

  T2: W(A)
  Commit

  T3: W(A)
  commit

  W(A)
  Commit

For view serializability, T1 needs to be first to satisfy cond 1. however, it will violate cond 3! See 2 for other cases.

1. The above is not view serializable (with any schedule: T1;T2;T3, T1;T3;T2, T2;T1;T3, T2:T3;T1, T3;T1;T2, T3:T2;T1)
2. Possible candidates are those ending with T1; but the read by T1 makes it not view serializable
3. However, it is serializble with a serial schedule whose last Tx is T1 As T1 write prevails

Example 3

- A view serializable schedule that is NOT conflict serializable

- S1: R1(A) W2(A) Commit2 W1(A) Commit1 W3(A) commit3

S1	S2
R1(A)	R1(A)
W2(A)	W1(A)
Commit2	Commit1
W1(A)	W2(A)
Commit1	Commit2
W3(A)	W3(A)
commit3	commit3

Cond i) A is read by T1 in both schedules
Cond ii) no read on a value written by another Tx
Cond iii) T3 performs the final write in both

Hence S1 is view serializable; it is view equivalent to S2 (a serial schedule)
However, S1 is NOT conflict serializable because two conflicting operations w2(a) w1(a) in S1 is not in the same order in S2

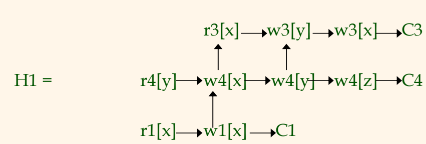
Serializability summary

- Conflict serializability is what is enforced by all DBMSs (by using locking protocols)
- Enforcing or testing view serializability turns out to be more difficult (computationally expensive and of little practical use).
- There are tests to determine view and conflict serializability
- But there is no simple test for serializability!!

Histories and serialization graph

T1 = r1[x] -> w1[x] -> C1
T3 = r3[x] -> w3[y] -> w3[x] -> C3
T4 = r4[y] -> w4[x] -> w4[y] -> w4[z] -> C4

A complete History over {T1, T3, T4} is



Conflict Serializable Histories

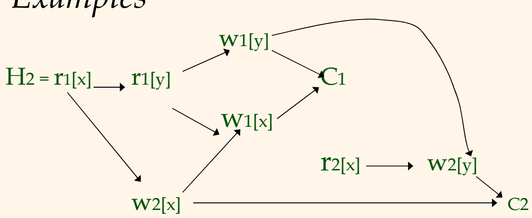
- Equivalence:

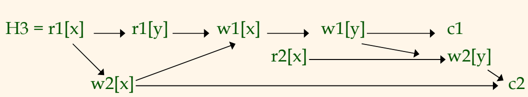
We define two histories H and H' to be equivalent (≡) if

(1) they are defined over the same set of transactions and have the same operations; and

(2) they order conflicting operations of non-aborted transactions in the same way; that is, for any conflicting operations p_i and q_j belonging to transactions T_i and T_j (respectively) where $a_i, a_j \notin H$, if $p_i <_H q_j$ then $p_i <_{H'} q_j$

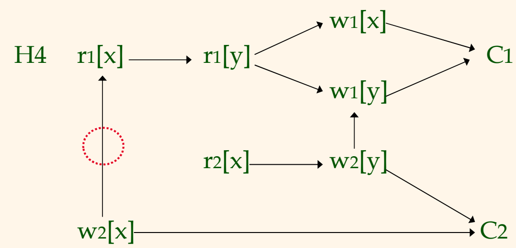
Examples





H2 and H3 are equivalent

Examples contd.

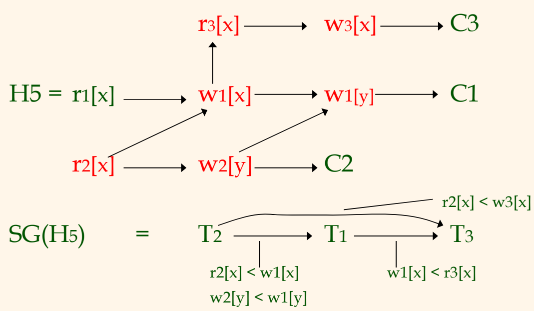


H4 is not equivalent to either H2 or H3

Serialization Graph(SG)

- Let H be a history over T = {T_1, T_2, ... T_n}. The serialization graph (SG) for H, denoted SG(H) is a directed graph whose nodes are the transactions in T that are committed in H and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. For example:

Serialization Graph(SG) contd.



The Serializability Theorem

- A history H is serializable (or conflict serializable based on the notion of conflicts) if SG(H) is acyclic.
- Proof sketch:
  - (if) Since SG(H) is acyclic, it can be topologically sorted. The sorted history will be equivalent to a serial history.
  - (only if) Serial history implies acyclicity.

Transitivity does not necessarily hold in a Serialization Graph(SG)

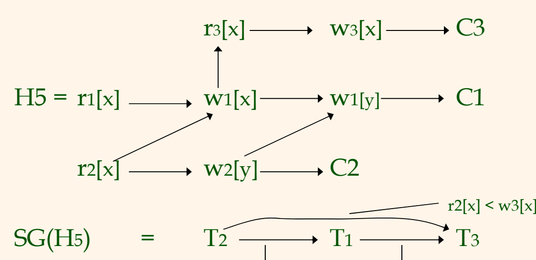
- In general, the existence of edges Ti → Tj and Tj → Tk in an SG does not necessarily imply the existence of edge Ti → Tk (Transitivity).

For example, with $w_3[z]$ replacing $w_3[x]$ in T_3, SG(H_5) becomes

$$T_2 \longrightarrow T_1 \longrightarrow T_3$$

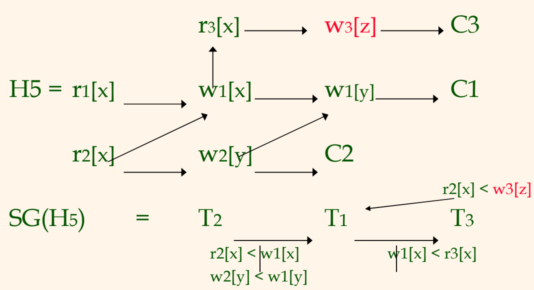
The transitive edge is not there.

Serialization Graph(SG) contd.



Transitivity does not
Necessarily hold!

Serialization Graph(SG) contd.



Recoverable Histories

- To ensure correctness in the presence of failures the schedule must produce executions that are not only serializable but also recoverable. Other desirable features are:
  - preventing cascading aborts
  - loss of before images
- Like serializability, recoverability can be conveniently formulated in terms of histories.

Cascading Aborts

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
Abort	
	Abort

- Is a situation where transaction T2 needs to be aborted because transaction T1 aborts
- This is necessary as T2 reads a value written by transaction T1
- If you want to avoid cascading aborts (ACA) such schedules should not be allowed!
- What is the easiest way to avoid such a situation?
- Postpone read until the Tx from which u want to read commits/aborts!

Recoverable Schedules

T1	***T2***
R(A)	
W(A)	
	R(A)
	W(A)
	commit
abort	

- Abort of T1 requires abort of T2!
  - But T2 has already committed!
- A recoverable schedule is one in which this cannot happen.
- What is the easiest way to avoid this?
  - i.e., a Xact commits only after all the Xacts it "depends on" (i.e., it reads from or overwrites) commit.
  - ACA implies Recoverable (but not vice-versa!).
- Real systems typically ensure that only recoverable schedules arise (through locking).
- Histories allow us to do serialization (conflict), ACA, and recoverability using one formalism!

Recoverable Histories contd.

- A transaction Ti reads data item x from Tj if Tj was the transaction that had last written into x but had not aborted at the time Ti reads x. More precisely, we say that Ti reads x from Tj in history H if
  - (1) wj[x] < ri[x] < means precedes (not less than!)
  - (2) a_j </ r_i[x] and
  - (3) if there is some w_k[x] such that W_j[x] < W_k[x] < r_i[x], then a_k < ri[x]
  - Note that a transaction can read a data item from itself. i.e., Wi[x] < ri[x]

Recoverable and ACA Histories

- A history is recoverable if each transaction commits after the commit of all transactions (other than itself) from which it read.
- A history H is called Recoverable (RC) if, whenever Ti reads from Tj (i ≠ j) in H and Ci ∈ H, Cj < Ci (or Ci waits till Cj commits).
- A history H avoids cascading aborts (ACA) if, whenever Ti reads x from Tj (i ≠ j), Cj < ri[x].
  - i.e., transactions may read only those values that are written by committed transactions or by itself.

Strict Histories

- A history H is strict (ST) if whenever wj[x] < Oi[x] (i ≠ j) , either a_j < Oi[x] or c_j < Oi[x], where Oi[x] is r_i[x] or w_i[x].
- That is, no data item may be read or overwritten until the transaction that previously written into it (note: not read by it) terminates, either by aborting or committing.
  - This is much stronger than the read by a Tx definition!

Examples

T1 = w1[x] w1[y] w1[z] C1
T2 = r2[u] w2[x] r2[y] w2[y] C2
H7 = w1[x] w1[y] r2[u] w2[x] r2[y] w2[y] C2 w1[z] C1
H8 = w1[x] w1[y] r2[u] w2[x] r2[y] w2[y] w1[z] C1 C2
H9 = w1[x] w1[y] r2[u] w2[x] w1[z] C1 r2[y] w2[y] C2
H_{10} = w1[x] w1[y] r2[u] w1[z] C1 W2[x] r2[y] w2[y] C2
H_{11} = w1[x] r2[y] w1[y] r2[u] w1[z] C1 W2[x] w2[y] C2 serializable?

H7 is not RC: T2 reads y from T1 but C2 < C1. Also not ACA
H8 is RC: T2 commits after T1 from which it read,
but not ACA: T2 reads y from T1 before T1 is committed.
H9 is ACA but not ST because T2 overwrites the value written into x by T1 before the latter terminates
H_{10} is ST

Theorem

$ST \subset ACA \subset RC$

i.e Recoverability, avoiding cascading aborts, and strictness are increasingly restrictive properties

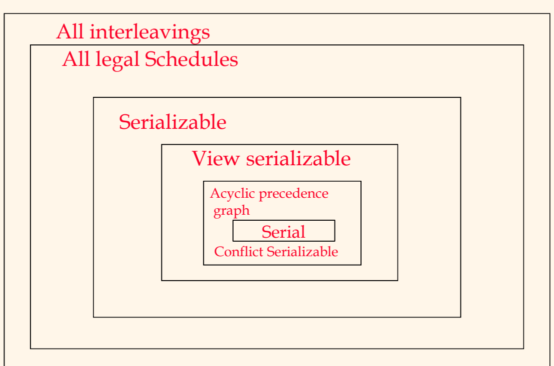
Proof:

Let $H \in ST$. Suppose T_i reads X from T_j in H ($i \neq j$). Then we have $w_j[X] < r_i[X]$ and $a_j \neg< r_i[X]$. Thus by definition of ST, $C_j < r_i[X]$

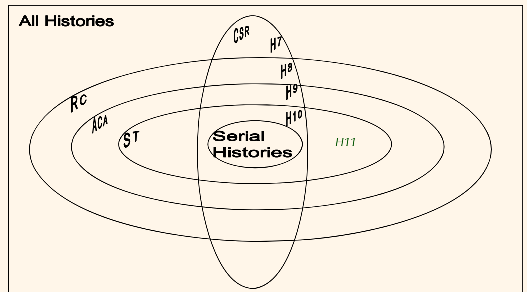
Hence $ST \subseteq ACA$. By the example shown $ST \neq ACA$. Hence $ST \subset ACA$.

Similarly, for $ACA \subset RC$

Venn Diagram of serializability



CSR intersects all of the sets RC, ACA and ST, but is incompatible with each of them. Two sets are incompatible if neither is contained in the other.



Venn Diagram illustrating relationships between histories

A ST history that is not CSR

H_{10} = w1[x] w1[y] r2[u] w1[z] C1 W2[x] r2[y] w2[y] C2

The above is ST and CSR (no cycles in the conflict graph)

H_{11} = w1[x] w1[y] r2[u] w1[z] W1[u] C1 W2[x] r2[y] w2[y] C2

Is ST, but NOT CSR

Why?

ST is satisfied if oi does Not follow w (but NOT read) without a or c

Summary of Concurrency control

- Concurrency control key to a DBMS.
  - More than just mutexes!
- Transactions and the ACID properties:
  - C & I are handled by concurrency control.
  - A & D coming soon with logging & recovery.
- Conflicts arise when two Xacts access the same object, and one of the Xacts is modifying it.
- Serial execution is our model of correctness.

Summary, cont.

- Serializability allows us to “simulate” serial execution with better performance.
- 2PL: A simple mechanism to get serializability.
  - Strict 2PL also gives us recoverability.
  - Conservative 2PL requests all locks at the beginning
- Lock manager automates 2PL so that only the access methods worry about it.
  - Lock table is a big main-memory hash table
- Deadlocks are possible, and typically a deadlock detector is used to solve the problem.

Thank You !