CSE 412/598
DATABASE MANAGEMENT
COURSE NOTES

11. TRANSACTIONS, RECOVERY & CONCURRENCY CONTROL

Department of Computer Science & Engineering
Arizona State University

TRANSACTION

an atomic (all or nothing) program unit that performs database access or update, taking a consistent (correct) database state into another consistent database state.

```
                                    database is consistent here
        begin transaction;/* Transfer funds */
                        balance1 ← balance1 - amount
                                    database is inconsistent here
                        balance2 ← balance2 + amount
        end transaction;    /* Transfer funds */
                                    database is consistent here
```

ACID properties of a transaction

Atomicity, Consistency, Isolation, Durability

ATOMICITY

"The system under test must guarantee that transactions are atomic; the system will either perform all individual operations on the data, or will assure that no partially-completed operations leave any effects on the data." [TPCA89]

ATM WITHDRAWAL (amt)

```
read bal
if bal > amt {
        bal ← bal - amt
        /* system goes down */
        dispense money
}
```

[TPCA89] TPC BENCHMARK™ A
Standard Specification, 10 November 1989

CONSISTENCY

"Consistency is the property that requires any execution of the transaction to take the database from one consistent state to another." [TPCA89]

Since the assumption is that a transaction is written so that it is consistent, consistency means that it remains consistent in the presence of the concurrent execution of transactions. (Serializability)

ATM WITHDRAWAL (amt1)

```
read bal
if bal > amt1
{
        bal ← bal - amt1
        dispense money
}
```

ATM WITHDRAWAL (amt2)

```
read bal
if bal > amt2
{
        bal ← bal - amt2
        dispense money
}
```

ISOLATION

The transaction should not reveal its uncommitted results to other transactions ...

ATM WITHDRAWAL (amt1)

```
read bal
if bal > amt1
{
        bal <- bal - amt1

        /* trans aborted */
        dispense money
}
```

ATM WITHDRAWAL (amt2)

```
read bal
if bal > amt2
{
        bal ← bal - amt2
        dispense money      }
```

DURABILITY

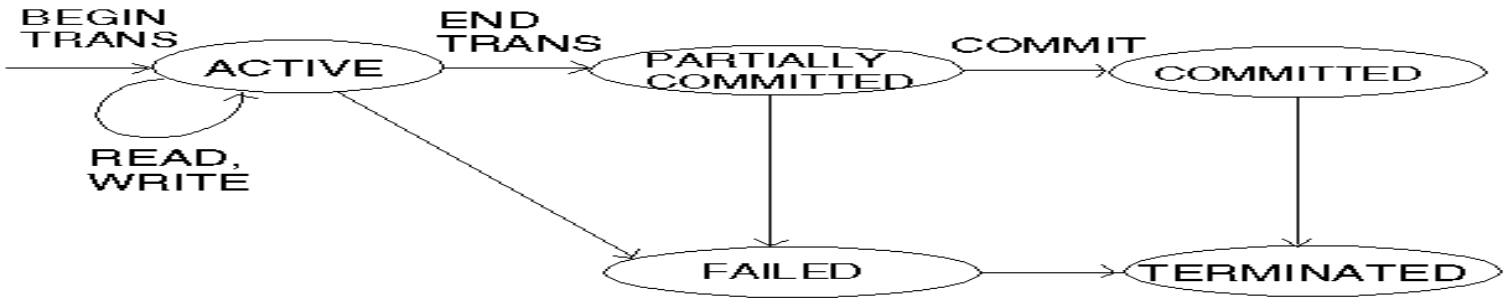
"The test bed system must guarantee the ability to preserve the effects of committed transactions and insure database consistency after recovery" from single failures... [TPCA89]

- permanent irrecoverable failure of any single durable medium containing database or recovery log data
- system crash/hang requiring system reboot
- memory failure (all or part)

RECOVERY CONTROL

integral part of the DB, responsible for

- the detection of failures
- restoration of the DB to a (consistent) state that existed prior to the failure.
- responsibility of the recovery scheme to ensure atomicity.



SYSTEM LOG

- Records all transaction operations that affect the value of database items to be able to recover from failure.
- Log is kept on disk & periodically backed up to archival storage.

Let T be a unique transaction ID,

X be a database item.

LOG ENTRIES

- [start-transaction, T]
- [write-item, T, X, new_value], or
  [write-item, T, X, old_value, new_value]
- [read-item, T, X]
- [commit, T]

SYSTEM LOG ASSUMPTIONS

- Transactions are not nested
- All permanent changes to the DB occur within transactions; so using the log, operations can be undone or redone based on transaction state.

INCREMENTAL LOG WITH IMMEDIATE UPDATES

This protocol uses

[start-transaction, T]

[read-item, T, X]

[write-item, T, X, old_value, new_value]

[commit, T]

Log records must be written to “stable storage” before executing an immediate update.

INCREMENTAL LOG WITH IMMEDIATE UPDATES

Recovery scheme:

Undo(T_i)	where T_i is uncommitted transaction
Redo(T_j)	where T_j is committed transaction

Redo & Undo must be idempotent – "executing it several times is equivalent to executing it once."

INCREMENTAL LOG WITH DEFERRED UPDATES

- During the execution of a transaction, all the write operations are deferred, recorded on a log and transaction workspace, until the transaction partially commits.
- At commit point, log is force written to disk and deferred writes are executed.

This protocol uses entries:

[start-transaction, T]

[write-item, T, X, new_value]

[commit, T]

INCREMENTAL LOG WITH DEFERRED UPDATES

Recovery scheme

- Log entries for uncommitted transactions are ignored because writes are deferred.
- Committed transactions must be redone (in order). Redo must be idempotent.

CHECKPOINTS

Log entry used to improve performance of recovery

[checkpoint] indicates that the system has written to the disk the effect of all write operations of committed transactions.

$\Rightarrow$ Transactions that have a commit entry before a checkpoint entry will **NOT** require their write operations to be redone in case of a system failure.

A checkpoint may contain additional info

- list of active IDs
- locations of first and most recent records in the log

SYSTEM LOG EXAMPLE

[start-trans T_1]

...

[commit T_1]

[start-trans T_2]

....

A:

[checkpoint]

B: ....

[commit T_2]

C:

SYSTEM LOG EXAMPLE
Recovery Scheme

	IMMEDIATE	DEFERRED
A	REDO T_1, UNDO T_2	REDO T_1
B	UNDO T_2	___
C	REDO T_2	REDO T_2

CONCURRENCY CONTROL

In a multiprogramming environment where several transactions may be executed concurrently, the system must control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

SCHEDULE:

an execution sequence that must preserve the order in which the instructions appear in each individual transaction

SERIAL SCHEDULE:

a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule (for a set of n transactions, there are n! legal serial schedules)

EXAMPLE SCHEDULES

T_0	T_1
read_item(A)	read_item(A)
write_item(A)	read_item(B)
read_item(B)	write_item(B)
write_item(B)	

equivalently:

$\mathbf{R_0(A)W_0(A)R_0(B)W_0(B)R_1(A)R_1(B)W_1(B)}$

Schedule of operations including multiple transactions.

S1:	$\mathbf{R_0(A)W_0(A)R_0(B)W_0(B)R_1(A)R_1(B)W_1(B)}$	-	serial
S2:	$\mathbf{R_1(A)R_1(B)W_1(B)R_0(A)W_0(A)R_0(B)W_0(B)}$	-	serial
S3:	$\mathbf{R_0(A)W_0(A)R_1(A)R_0(B)W_0(A)R_1(B)W_1(B)}$	-	not serial

Is it equivalent to a serial schedule?

Yes, it is equivalent to S1 $\Rightarrow$ “serializability”

SERIALIZABILITY

ensuring that the outcome of processing a set of transactions concurrently be the same as one produced by running these transactions serially in some order

***COMPUTATIONALLY EQUIVALENT*:**

$S_1 = S_2$ if

1. The set of transactions that participate in S_1 and S_2 are the same.
2. For each data item X, if in S_1 trans T_i executes read(X) and the value of X read by T_i was written by T_j, then the same will hold in S_2.
3. For each data item X, if in S_1 trans T_i executes the last write(X) instruction, then the same holds in S_2.

S is SERIALIZABLE if there is a serial schedule S' s.t. S=S'.

PRECEDENCE GRAPH

a direct graph G = (V,E) where

V = set of transactions participating in the schedule

E = set of edges $T_i \rightarrow T_j$ for which one of the following holds:

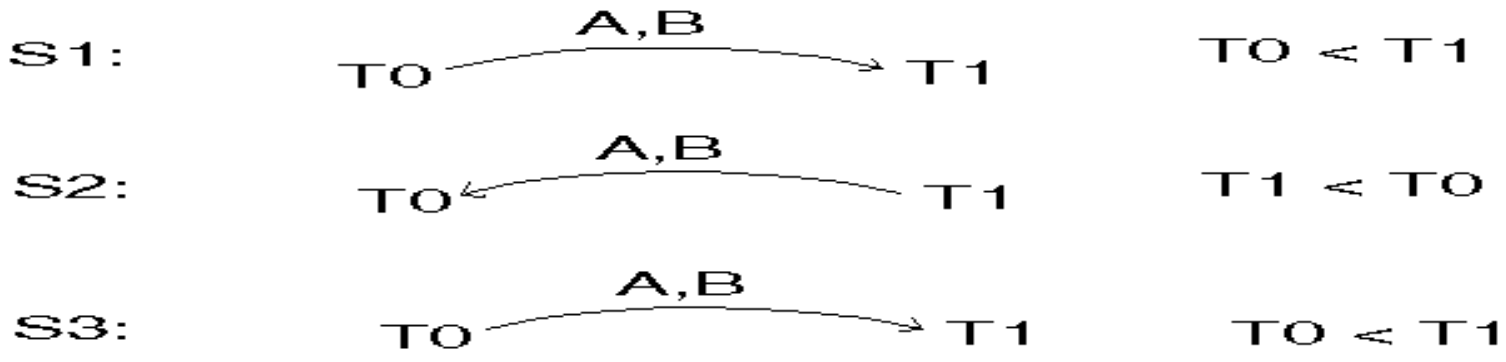
1. T_i executes write(X) before T_j executes read(X)
   (only when T_j reads value written by T_i)
2. T_i executes read(X) before T_j executes write(X)
   (Intuitively, an edge $T_i \rightarrow T_j$ implies that in any serial schedule S' equivalent to S, T_i must appear before T_j.)

***CONSTRAINED WRITE ASSUMPTION*:**

A transaction must read a data item before it can write it.

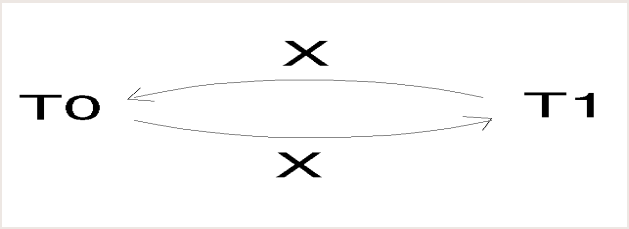
The schedule S is serializable if the precedence graph for S contains no cycles.

SERIALIZABILITY EXAMPLES



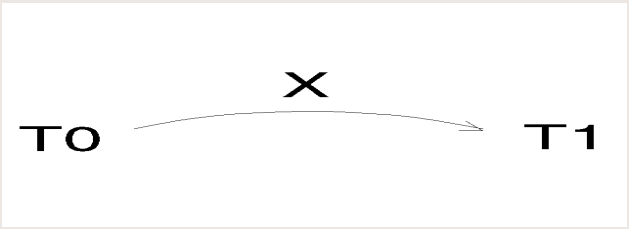
MORE SERIALIZABILITY EXAMPLES

S_C: **$R_0(X)R_1(X)W_0(X)R_0(Y)W_1(X)W_0(Y)$**



not serializable

S_D: **$R_0(X)W_0(X)R_1(X)W_1(X)R_0(Y)W_0(Y)$**



serializable

equivalent to serial schedule

$R_0(X)W_0(X)R_0(Y)W_0(Y)R_1(X)W_1(X)$

SERIALIZABILITY ORDER

a linear order consistent with the partial order of the precedence graph

Impractical to test for serializability ... protocols (sets of rules) are used to ensure serializability

- LOCKING (2PL)
- TIMESTAMPS

All schedules allowed by 2PL/timestamps are serializable,
but 2PL/timestamps do not allow all possible serializable schedules.

LOCKING

To ensure serializability require access by locks.

LOCK MODES

READ/SHARED:

if a trans T has obtained a shared mode lock on item X by *read_lock(X)/LS(X)*, then T can read this item but it cannot write X

WRITE/EXCLUSIVE:

if a trans T has obtained an exclusive mode lock on item X by *write_lock(X)/LX(X)*, then T can both read and write X

UNLOCKED:

T unlocks data item X by issuing an *unlock_item(X)/UN(X)*

LOCK COMPATIBILITY MATRIX

	READ REQUEST	WRITE REQUEST
READ LOCK	LOCK	WAIT
WRITE LOCK	WAIT	WAIT

LOCK:

Lock manager issues lock to requesting transaction

WAIT:

Transaction requesting lock must enter a wait state until lock issued by lock manager.

LOCKING ASSUMPTIONS

Well-formed Assumption:

Every transaction requests a lock in an appropriate mode on data item X depending on the type of operations it will perform on X. (Therefore, locks will not be escalated.) If the data item is already locked by another transaction in an incompatible mode, then T must wait until all incompatible locks held by other transactions have been released.

LOCKING OVERVIEW

1. If a trans unlocks a data item immediately after its last access of the data item, serializability may not be ensured.
2. Two-phase locking protocol (2PL) - ensures serializability.
   - Growing phase: a trans may obtain locks but may not release any locks.
   - Shrinking phase: a trans may release locks but may not obtain any new locks.
3. To guarantee isolation, locks must be held until the transaction reaches its commit point.
4. Deadlock can occur - each of two transactions is waiting for the other to release an item.
5. Livelock can occur - a transaction cannot proceed for an indefinite amount of time while other transactions continue normally (lock manager must be fair).

EXAMPLE TRANSACTIONS

T_1:	Display A+B	T_2:	Inc A & Dec B
	R(B)		R(A)
	R(A)		$A \leftarrow A + 1$
	Display A+B		W(A)
			R(B)
			$B \leftarrow B - 1$
			W(B)

WELL-FORMED BUT NOT SERIALIAZABLE

T_1	T_2
LS(B)	
R(B)	
UN(B)	
	LX(A)
	R(A)
	A ← A + 1
	W(A)
	UN(A)
	LX(B)
	R(B)
	B ← B – 1
	W(B)
	UN(B)
LS(A)	
R(A)	
UN(A)	
Display A+B	

S: $R_1(B)R_2(A)W_2(A)R_2(B)W_2(B)R_1(A)$

2PL: SERIALIAZABLE BUT NOT ISOLATED

T_1'	T_2
	LX(A)
	R(A)
	A ← A + 1
	W(A)
	LX(B)
	UN(A)
LS(A)	
R(A)	
	R(B)
	B ← B – 1
	W(B)
	UN(B)
LS(B)	
UN(A)	
R(B)	
UN(B)	
Display A+B	

S: $R_2(A)W_2(A)R_1(A)R_2(B)W_2(B)R_1(B)$

2PL & ISOLATION

T_1	T_2	T_3
LS(B)		
R(B)		
		LS(B)
LS(A)		
R(A)		
		R(B)
		UN(B)
UN(A)		
	LX(A)	
	R(A)	
	A ← A + 1	
	W(A)	
UN(B)		
	LX(B)	
	R(B)	
	B ← B – 1	
	W(B)	
	UN(B)	
	UN(A)	

S: $R_1(B)R_1(A)R_3(B)R_2(A)W_2(A)R_2(B)W_2(B)$

DEADLOCK

T_1	T_2
LS(B)	
R(B)	
	LX(A)
	R(A)
	A ← A + 1
	W(A)
LS(A)	
	LX(B)

DEADLOCK

DEADLOCK - waiting for an event that will not occur

Necessary conditions for a deadlock to exist:

- mutual exclusion: exclusive control of resources
- wait for: resources held while waiting for additional ones
- no preemption: resources are held until used to completion
- circular wait: each transaction holds one or more resources requested by the next transaction in the circular chain

DEADLOCK APPROACHES

- TIMEOUT: abort a transaction after it has been in a wait state for a specified time interval
  - may abort a transaction not in deadlock
  - difficult to determine appropriate timeout interval
  - cascading effect due to system overload
- DEADLOCK PREVENTION: a transaction is not allowed to enter a wait state if there is a risk of deadlock
- DEADLOCK DETECTION: periodically check if the system is in deadlock and rollback a deadlocked transaction

TIMESTAMPS

A timestamp is a unique identifier assigned to a transaction when it starts. Concurrency control techniques use the timestamps to guarantee serializability. A transaction is aborted and restarted if the execution of its next operation would violate the serializability order. Thus, concurrency control based on timestamps do not suffer from deadlock.

TIMESTAMP ORDERING:

selecting a serializability order in advance between every pair of trans [not exactly - see protocol]

TIMESTAMPS

Implementation

$TS(T_i)$: unique fixed timestamp for trans T_i; assigned by the db system before Ti starts execution

If a trans T_i has timestamp $TS(T_i)$ and a new trans T_j enters the system, then $TS(T_i) < TS(T_j)$

Implementation of timestamp:

1) Use the value of the system clock
2) Use a logical counter that is incremented after a new timestamp has been assigned

write_TS(X):

the largest timestamp of any trans that successfully executes write(X)

read_TS(X):

the largest timestamp of any trans that successfully executes read(X)

BASIC TIMESTAMP METHOD

Let TS: timestamp of transaction requesting operation

READ(x,TS)

```
    if TS < write_TS(x)
            reject & restart
    else
            execute read
            read_TS(x) ← max(read_TS(x), TS)
```

WRITE(x,TS) /* assume read before write */

```
    if TS < read_TS(x)
            reject & restart
    else
            execute write
            write_TS ← TS
```

NOTES ON TIMESTAMPS

1) A trans Ti, which is rolled back as a result of the concurrency control scheme, is assigned a unique timestamp and restarted
2) Serializability ensured
3) Freedom from deadlock (since no trans ever waits)
4) May result in cascading rollbacks and livelock (“cyclic restart”)

NEITHER LOCKING NOR TIMESTAMPS ALLOW ALL POSSIBLE SERIALIZABLE SCHEDULES...

some schedules are possible under each protocol that are not allowed in the other

TIMESTAMPS OK - NOT 2PL

T_0	T_1	T_2
LX(A)		
LX(B)		
R-W(A)		
UN(A)		
	LX(A)	
	R-W(A)	
	UN(A)	
		LX(A)
		R-W(A)
		UN(A)
R-W(B)		
UN(B)		
	R-W(B) ← cannot get lock on B	

2PL OK - NOT TIMESTAMPS

T_0	T_1
LS(A)	
R(A)	
	LX(B)
	R-W(B)
	UN(B)
LS(B)	
R(B) ← write-TS(B) = 1 & 0 < 1, ∴ reject & restart	
UN(A)	
UN(B)	