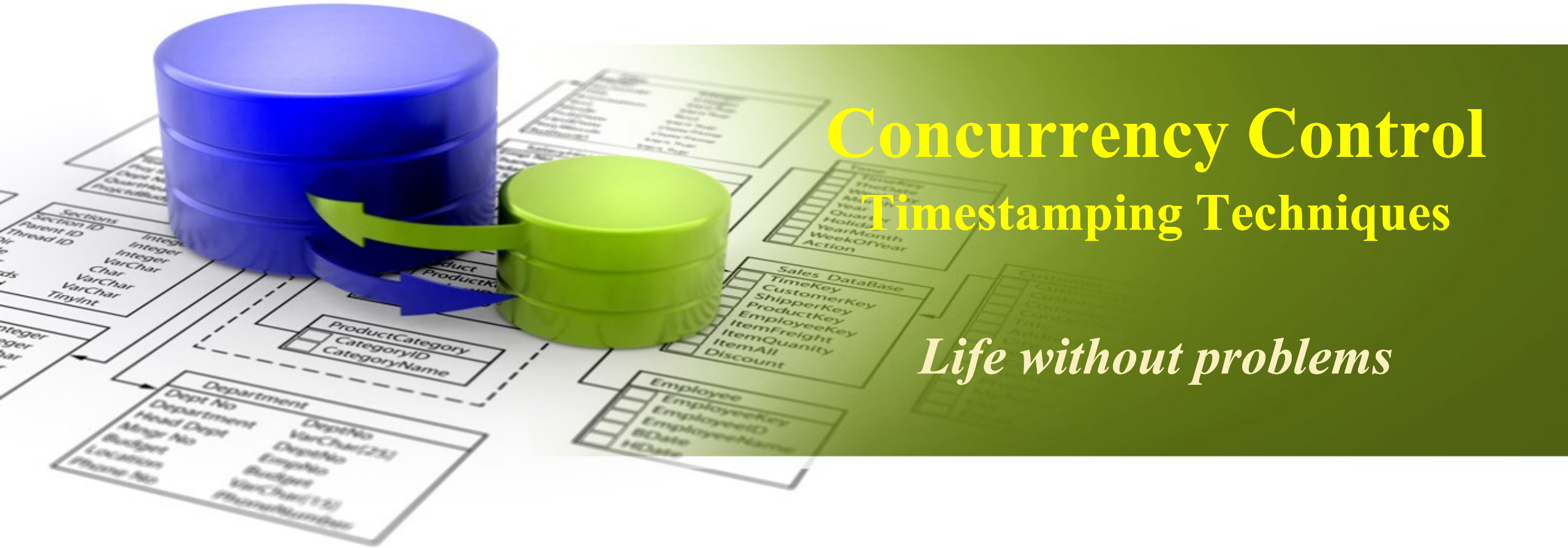
CS 422 - DBMS

Lesson 11 - Chapter 22



Concurrency Control Techniques

- Two basic concurrency control techniques (for achieving Serializability):
  - **Locking**
  - **Timestamping**
- Both are *conservative* (*pessimistic*) approaches:
  - delay transactions in case they conflict with other transactions some time in future.
- Optimistic methods are available which assume conflict is rare and only check for conflicts at commit.
  - They allow transactions to proceed unsynchronized and check for conflicts only at the end, when a transaction commits.

Granularity of Data Items

- It is the **size of data items** chosen as **unit of protection** by concurrency control protocol.
- The size or granularity of the data item that can be locked in a single operation has a significant effect on the overall performance of the concurrency control algorithm.

Granularity of Data Items contd..

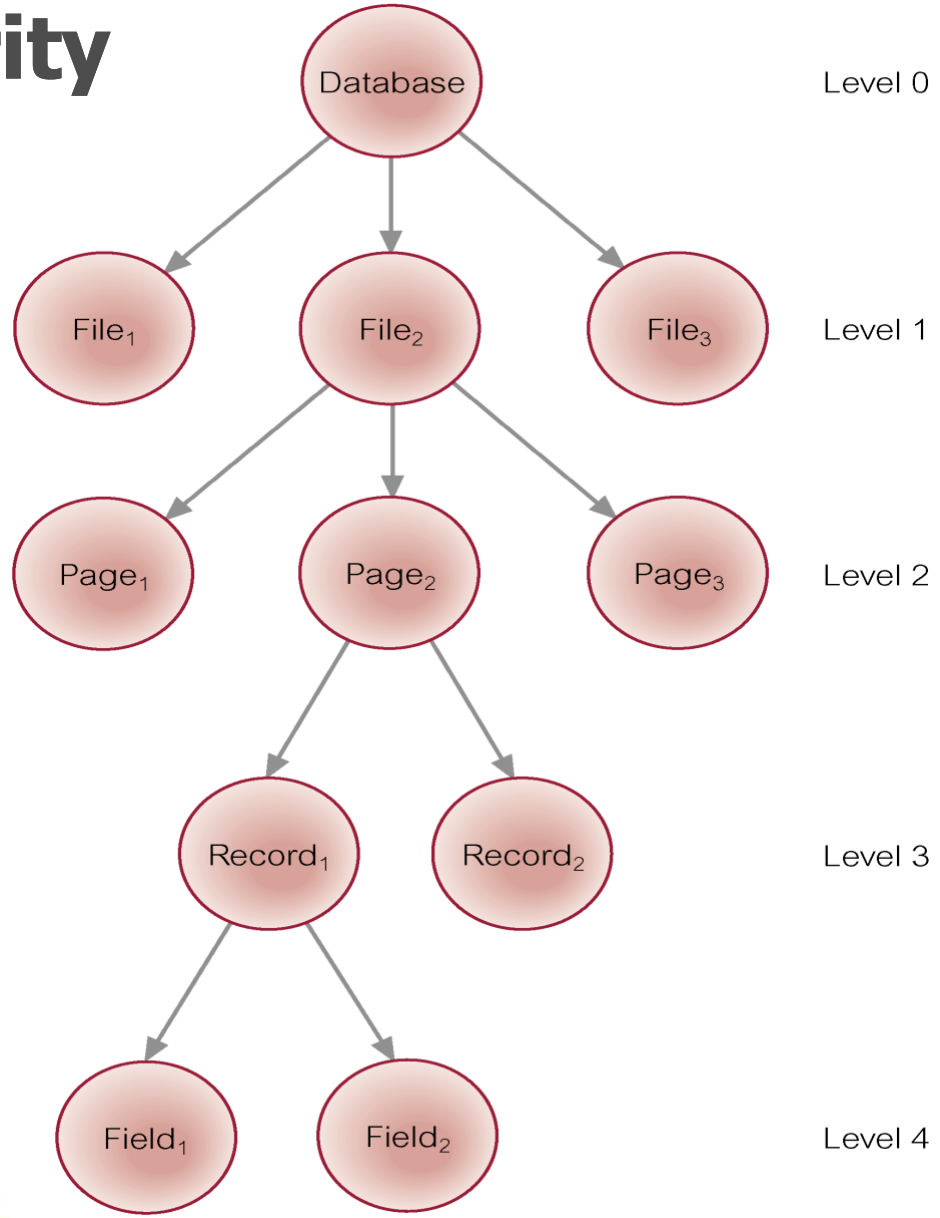
- **Data item is chosen to be one of the following ranging from coarse to fine:**
  - The entire database
  - A file/table
  - A page (or area or database space)
    - A section on physical disk in which relations are stored.
  - A record/row
  - A field value of a record

Granularity of Data Items contd..

- **Tradeoff:**
  - The coarser the data item size, the lower the degree of concurrency permitted.
  - The finer the item size, the more locking information that needs to be stored.
- **Best item size depends on the nature of the transactions.**
- If a typical transaction accesses a small number of rows, it is advantageous to have the data item granularity at the row level.
- On the other hand, if a transaction typically accesses many rows of the same table, it may be better to have page or table granularity so that the transaction considers all those records as one (or a few) data items.

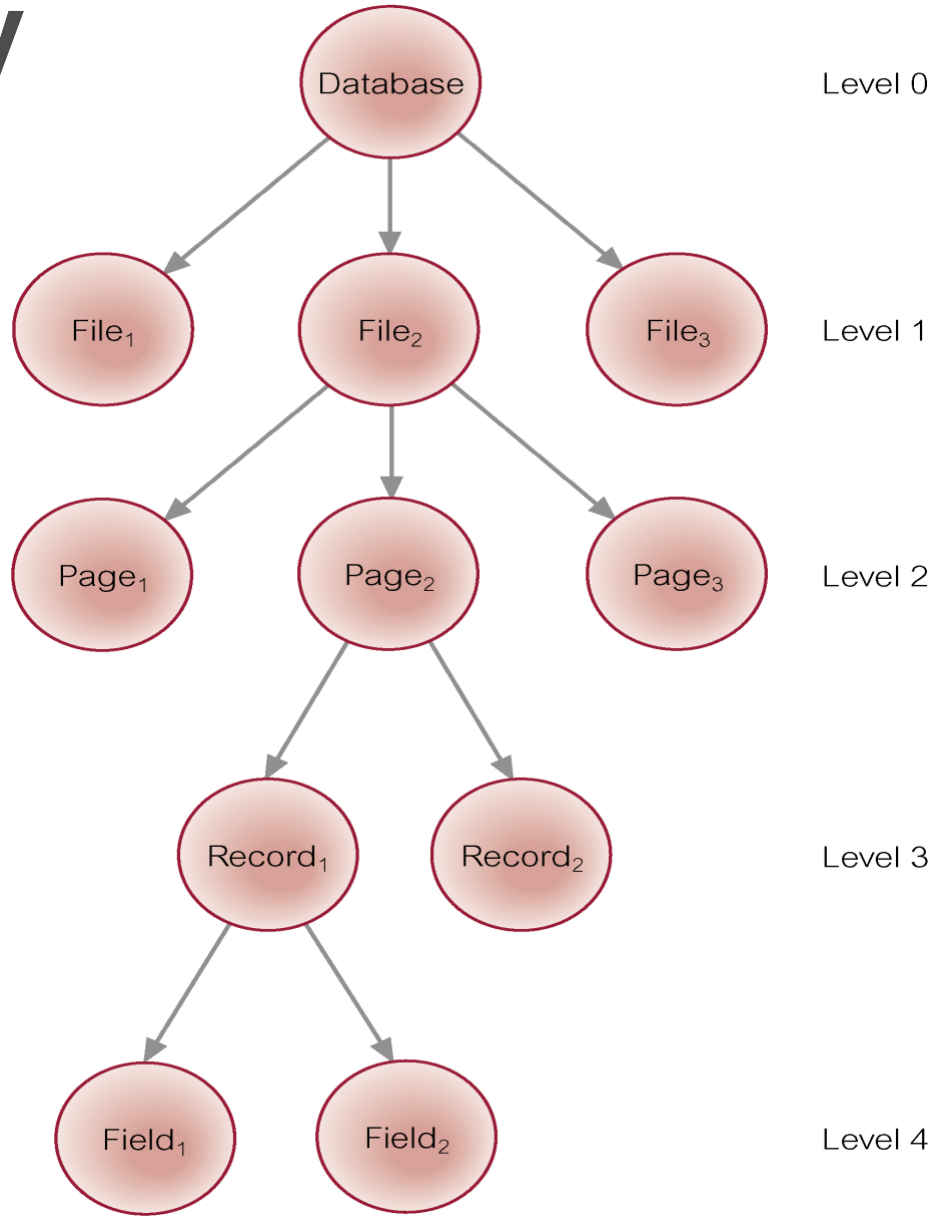
Hierarchy of Granularity (Levels of locking)

- Could represent granularity of locks in a hierarchical structure where each node represents data items of different sizes.
- Root node represents entire database, level 1s represent files, etc.
- Whenever a node is locked, all its descendants are also locked.
- DBMS checks the hierarchical path before granting locks.



Hierarchy of Granularity (Levels of locking) contd..

- If a transaction locks Page2, all its records (Record1 & Record2) as well as all their fields (Field1 & Field2) are also locked.
- If a request is for an exclusive lock on *Record1*, DBMS checks its parent (*Page2*), its grandparent (*File2*), and the database itself to determine whether any of them are locked.
- When it finds that *Page2* is already locked, it denies the request.
- If a lock is requested on *File2*, DBMS checks every page in the file, every record in those pages, and every field in those records to determine whether any of them are locked.



Multiple Granularity Locking

- This specialized locking strategy is used by DBMS to reduce the searching involved in locating locks on descendants.
- *Intention lock* could be used to lock all ancestors of a locked node.
- Thus, if some descendant of $File_2$ (in our example, $Page_2$) is locked and a request is made for a lock on $File_2$, the presence of an intention lock on $File_2$ indicates that some descendant of that node is already locked.
- Intention locks can be *read (shared)* or *write (exclusive)*. Applied top-down, released bottom-up.

Multiple Granularity Locking
Types of Locks

- **Shared (S) :** Read lock
- **Exclusive (X) :** Write lock
- **Intention-Shared (IS):** At least one descendant has a shared (S) lock.
- **Intention-Exclusive (IX):** At least one descendant has an exclusive (X) lock.
- **Shared + Intention-Exclusive (SIX):** The node by itself is locked by S lock and at least one descendant has an X lock.

Multiple Granularity Locking
Locking Rules

- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- To ensure Serializability with locking, a 2PL protocol is used.
- A transaction may obtain only the most restrictive lock on one database item at a time. (Meaning, if a transaction wants S and X locks on data item A then the most restrictive lock X will be granted.)
- No node may be locked until it's parent is locked by an intention lock.
- No node may be unlocked until all its descendants are unlocked.
- No lock may be granted once any node has been unlocked.
- Locks are acquired in a root to leaf order and locks are released in a leaf to root order.

Multiple Granularity Locking
Lock Compatibility Matrix

	IS	IX	S	SIX	X
IS	✔	✔	✔	✔	✗
IX	✔	✔	✗	✗	✗
S	✔	✗	✔	✗	✗
SIX	✔	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

IS : Intention Shared
IX : Intention Exclusive
S : Shared
X : Exclusive
SIX : Shared & Intention Exclusive

Timestamping

- A concurrency control protocol that also guarantees Serializability.
- Uses transaction timestamps to order transaction execution for an equivalent serial schedule.
- Transactions are ordered in such a way that older transactions (transactions with *smaller* timestamps), get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting transaction.

Difference between Locking & Timestamping Methods for Concurrency Control

- Timestamp methods for concurrency control are quite different from locking methods.
- No locks are involved and therefore there can be no deadlock.
- Locking methods generally prevent conflicts by making transactions wait. With timestamp methods, there is no waiting: transactions involved in conflict are simply rolled back and restarted.

Timestamping

Timestamp

A unique identifier created by DBMS that indicates relative starting time of a transaction.

- Can be generated by using system clock at the time when transaction started, or by incrementing a logical counter every time a new transaction starts.

Timestamping Rules

- Read/write proceeds only if *last update on that data item* was NOT carried out by a younger transaction.
- Otherwise, the current transaction requesting read/write is aborted and restarted and given a *new timestamp*.
- Data items are also assigned timestamps :
  - **read-timestamp** - timestamp of last transaction who read this data item;
  - **write-timestamp** - timestamp of last transaction who wrote to this data item.

Timestamping Protocols

- **Single Version Timestamping**
  (Basic timestamping)
- **Multiversion Timestamping**

Single-Version Timestamping - Read(x)

- Consider a transaction T with timestamp ts(T) and it issues a read request on data item x, then the single version timestamping protocol works as follows:

  If ts(T) < write_timestamp(x)

  - x already updated by younger (later) transaction.
  - This transaction T is too late to read the previous outdated value, and any other values it has acquired are likely to be inconsistent with the updated value of the data item x.
  - So transaction T must be aborted and restarted with a new timestamp.

- Otherwise if **ts(T) >= write_timestamp(x)**, read can proceed.

  - **read_timestamp(x) = max {ts(T), read_timestamp(x)}**

Single-Version Timestamping - Write(x)

- The same transaction T with timestamp ts(T) issues a write request on data item x:

If ts(T) < read_timestamp(x)

- x already read by younger transaction.
- Means, a later transaction is already using the current value of the item and it would be an error to update it now. This occurs when a transaction is late in doing a write and a younger transaction has already read the old value or written a new one.
- So roll back this requesting transaction T and restart it using a new timestamp.

If ts(T) < write_timestamp(x)

- x already written by younger transaction.
- Means, transaction T is attempting to write an obsolete value of data item x.
- So roll back this requesting transaction T and restart it using a new timestamp.

- Otherwise, operation is accepted and executed.

write_timestamp(x) = ts(T)

Thomas's Write Rule

Ignore obsolete write rule – For greater concurrency

- Consider the same transaction T issues a write request on data item x:

  ts(T) < read_timestamp(x)

  - x already read by younger transaction.
  - So roll back this requesting transaction T and restart it using a new timestamp.

  ts(T) < write_timestamp(x)

  - x already written by younger transaction.
  - Write can safely be ignored - *ignore obsolete write* rule.

- Otherwise, operation is accepted and executed.

  - **write_timestamp(x) = ts(T)**

Single Version Timestamping Rules

Read(X)

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

Abort, Rollback, Restart

Else

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

Write(X)

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

Abort, Rollback, Restart

Else If X is **written** by a younger transaction

Ignore Write // Thomas's write rule

Else

Write // set write_ts as ts(T)

Examine the Use of Thomas's Write Rule

Read(X)
- If X is **written** by a younger transaction
  - Abort, Rollback, Restart
- Else
  - Read //read_ts = max{ts(T), read_ts}

Write(X)
- If X is **read** by a younger transaction
  - Abort, Rollback, Restart
- Else If X is **written** by a younger transaction
  - Ignore Write // Thomas's write rule
- Else
  - Write // set write_ts as ts(T)

Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read(bal_x)		
t_3		begin_transaction	
t_4		write(bal_x)	
t_5		commit	
t_6	write(bal_x)		
t_7	commit		
t_8			begin_transaction
t_9			write(bal_x)
t_{10}			commit

Single Version Timestamping Example

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

Read(X)
If X is **written** by a younger transaction
Abort, Rollback, Restart
Else
Read //read_ts = max{ts(T), read_ts}

Write(X)
If X is **read** by a younger transaction
Abort, Rollback, Restart
Else If X is **written** by a younger transaction
Ignore Write
Else
Write // set write_ts as ts(T)

S operations	(Variable(readTS, writeTS))
(t1, START (T1))	
(t2, read(T1, X))	X(t1,_)
(t3, START (T2))	
(t4, read(T2, Y))	Y(t3, _)
(t5, START (T3))	
(t6, write(T3, X))	X(t1, t5)
(t7, read(T2, X))	TS(T2) = t3 < writeTS(X) = t5, Abort, Roll back, Restart T2
(t8, read(T1, Y))	Y(t3, _) t3 = max{t3, t1}
(t9, commit(T1))	
(t10, START(T2))	
(t11, commit(T3))	
(t12, read(T2, Y))	Y(t10, _) t10 = max{t3, t10}
(t13, read(T2, X))	TS(T2) = t10 >= writeTS(X) = t5, X(t10, t5) //t10 = max{t1, t10}
(t14, commit(T2))	

Another Example – Basic Timestamp Ordering

Read(X)
If X is **written** by a younger transaction
Abort, Rollback, Restart
Else
Read //read_ts = max{ts(T), read_ts}

Write(X)
If X is **read** by a younger transaction
Abort, Rollback, Restart
Else If X is **written** by a younger transaction
Ignore Write
Else
Write // set write_ts as ts(T)

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)†	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)‡	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

† At time t_8, the write by transaction T_{20} violates the first timestamping write rule described previously and therefore is aborted and restarted at time t_{14}.
‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Multi-version Timestamp Ordering

- Basic timestamp ordering protocol assumes only one version of data item exists, and so only one transaction can access data item at a time.
- Multi-version timestamp ordering protocol uses versioning of data to increase concurrency.
- Can allow multiple transactions to read and write different versions of same data item, and ensure each transaction sees consistent set of versions for all data items it accesses.

Multi-version Timestamp Ordering

- In Multiversion timestamp concurrency control, **each write operation creates new version of data item** while retaining old version.
- When transaction attempts to read data item, system selects the correct version of the data item according to the timestamp of the requesting transaction.
- Versions can be deleted once they are no longer required.

Multi-version Timestamps

- For each data item x, the database holds **n** versions x_1, x_2, …, x_n.
- For each version i, system stores:
  - the value of version x_i
  - **read_timestamp(x_i)** –largest timestamp of transaction that have read version x_i;
  - **write_timestamp(x_i)** –timestamp of transaction that created version x_i.

Multi-version Timestamp Rules

- Let ts(T) be the timestamp of current transaction T.
- **Transaction T issues a write(*x*) request:**
  - if x_i has the largest write timestamp of *x* that is less than or equal to ts(T) and
    - if read_timestamp(x_i) <= ts(T) then create a new version of *x.*
    - else if read_timestamp(x_i) > ts(T) then transaction T is aborted & restarted with new timestamp.
- **Transaction T issues a read(x) request:**
  - return the version x_i that has largest write timestamp of *x* that is less than or equal to ts(T).
  - set read_timestamp(x_i) = max (ts(T), read_timestamp(x_i))
  - read operation never fails

Multi-Version Timestamp Rules

Read(X)

Pick the version.

The version with largest write_ts <= ts(T)

Read // Read never fails

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

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 timestamp as ts(T)

Else

Abort, Rollback, Restart

Multi Version Timestamping Example

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

Read(X)
  Pick the version.
    The version with largest write_ts <= ts(T)
      Read // Read never fails
            //read_ts = max{ts(T), read_ts}

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 timestamp as ts(T)
    Else
        Abort, Rollback, Restart

Operations of S	$VAR_{version}$(readTS, writeTS))
(t_1, START (T1))	// Assume $X_0(t_0, t_0)$, $Yo(t_0, t_0)$
(t2, read(T1, X))	$X_0(t_1, t_0)$
(t_3, START (T2))	
(t4, read(T2, Y))	$Y_0(t_3, t_0)$
(t_5, START (T3))	
(t6, write(T3, X))	$X_1(t_5, t_5)$
(t7, read(T2, X))	// X_0 selected $X_0(t_3, t_0)$
(t8, read(T1, Y))	$Y_0(t_3, t_0)$ $t3 = max\{t_3, t_1\}$
(t9, commit(T1))	
(t10, commit(T2))	
(t11, commit(T3))	

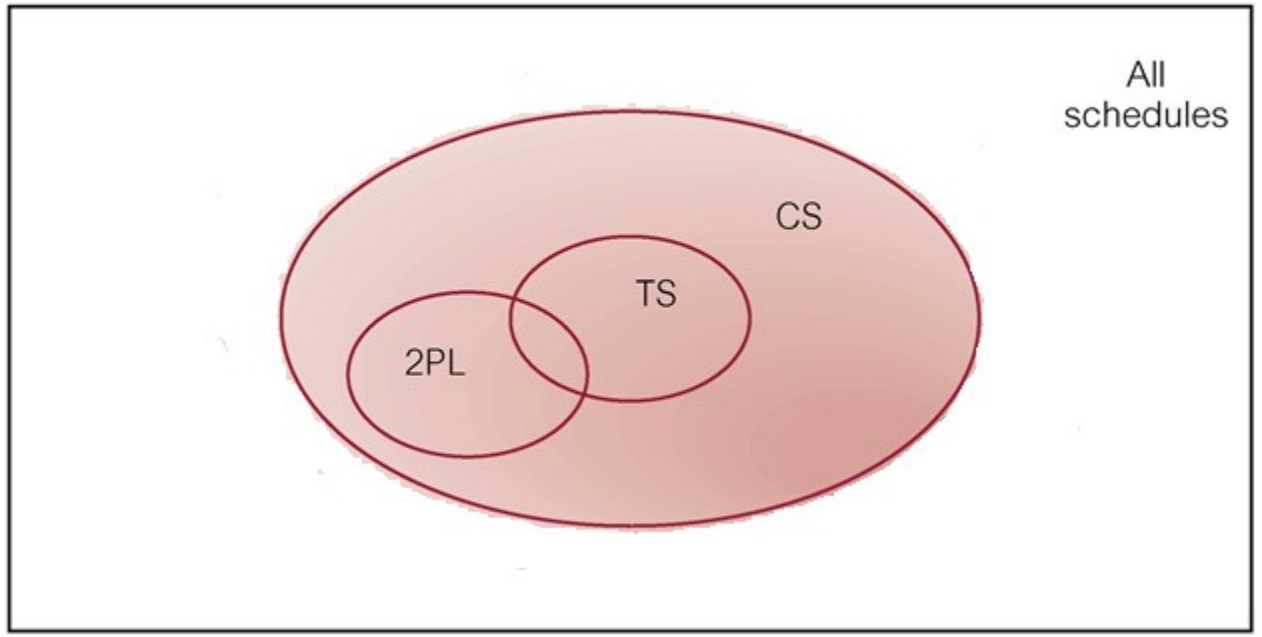
In-Class Exercises

Suppose our DBMS uses timestamp concurrency control protocol.

How is the following set of transaction requests will be handled by MVCC?

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

Comparison of Methods



CS - Conflict-Serializable schedules
TS – Schedules created by Timestamping methods
2PL – Schedules created by 2PL