Timestamping algorithm

Topic 6, Lesson 6
An alternative to a locking protocol for concurrency control

Timestamping Algorithm

Concurrency control solution: alternative to a locking protocol, no locks so no deadlocks

Borrow an idea from the deadlock prevention algorithms: place an order on the transactions, use the time order of the transactions to decide what operations are allowed in the database.

Conflict is resolved by rolling back and restarting transaction. Older transactions, transactions with **smaller** timestamps, get less of a priority in the event of conflict.

Timestamp for transactions

Each transaction is assigned a unique timestamp

A timestamp is a unique identifier created by DBMS that indicates relative starting time of a transaction.
Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.

Transaction's timestamp will indicate the order that the conflicting operations in the transactions are executed.

Timestamps for data objects

Also timestamps for data items:

- **read-timestamp** - timestamp of last (most recent) transaction to read item
- **write-timestamp** - timestamp of last (most recent) transaction to write item

When performing I/O operations, the read/write proceeds only if **last update on that data item** was carried out by an older transaction.

Otherwise, transaction requesting read/write is restarted and given a new timestamp, since it is considered stale

Timestamping algorithm

Two versions:

Basic timestamp ordering (BTO) algorithm – ensures that the timestamp for the transactions determines the order that the conflicting operations are completed (conflict serializable)

Thomas write rule (TW) algorithm – provides an optimization for dealing with stale updates of a data object. TW identifies when stale updates can be safely ignored and not completed

Neither version guarantees a recoverable schedule

Timestamping Algorithm: Read(x)

Consider a transaction T with timestamp ts(T):

ts(T) < write_timestamp(x)
x already updated by younger (later) transaction.
Transaction must be aborted and restarted with a new timestamp.

Otherwise ts(T) > write_timestamp(x)
operation is accepted and executed.
Update the read_timestamp(x)
read_timestamp = max((ts(T), read_timestamp(x))

Timestamping Algorithm: write(x) BTO

ts(T) < read_timestamp(x)

x already read by a younger transaction.
Roll back transaction and restart it using a later timestamp.

ts(T) < write_timestamp(x)

x already written by younger transaction. T's version of x is obsolete.
Roll back transaction and restart it using a later timestamp.

Otherwise, operation is accepted and executed. Set the **write_stamp = ts(T)**

Timestamping Algorithm: write(x) (TW)

ts(T) < read_timestamp(x)

x already read by a younger transaction.
Roll back transaction and restart it using a later timestamp.

ts(T) < write_timestamp(x)

x already written by younger transaction. T's version of x is obsolete.
Write can safely be ignored - **ignore obsolete write Thomas' write** rule.

Otherwise, operation is accepted and executed. Set the **write_stamp = ts(T)**

TW rule: Simple Example

Time	Transaction 1	Transaction 2
t_1	Start Transaction	
t_2		Start Transaction
t_3		Read(A)
t_4	Read(B)	
t_5		Write(C)
t_6	**Write(C)**	
t_7	Commit	
t_8		Commit

Ignored

Time	Transaction 1	Transaction 2
t_1	Start Transaction	
t_2		Start Transaction
t_3		Read(A)
t_4	Read(B)	
t_5		Write(C)
t_6		
t_7	Commit	
t_8		Commit

Timestamping algorithm: Example

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}	$bal_z = 50$	$bal_z = 50$	T_{22}	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 by ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Disadvantages to timestamping algorithm

If a transaction is aborted, it is restarted with a new timestamp. This can result in a cyclic restart where a transaction can **repeatedly restart and abort without ever completing,** penalizes long-running transactions.

Another disadvantage is that it has storage overhead for maintaining timestamps (two timestamps must be kept for every data object).

Also requires more write operations to the database since we are tracking and updating each access (read or write) to the database objects.

Summary

- Timestamp is another method for supporting concurrency
- It is a simpler solution to locking since it only requires tracking read and write access to data objects and placing an ordering on the transactions
- It can suffer from starvation of long-running transactions
- It requires more updates to the database, since it does an update of the read timestamp when transaction's read the data.