Advanced Databases

Concurrency Control Techniques II

Dr. George Mertzios
Michaelmas Term

george.mertzios@durham.ac.uk
Room 2066, MCS Building
Tel: 42 429

Recoverable schedules

- Recoverable schedule: a schedule that ensures:
  - once a transaction *T* is committed,
  - it should never be necessary to roll back *T*

- Recoverable schedules guarantee durability of transactions :
  - “the effects of a committed transaction are permanently recorded”

  (*recall:* ACI**D** properties)

Recoverable schedules

- A transaction *T* reads from transaction *T'* in a schedule *S*, if some data item x is:
  - first written by *T'* and
  - then read by *T*

- A schedule S is recoverable if no transaction *T* commits in S, unless:
  - first all transactions *T'* commit, from which *T* reads

- If T_9 commits immediately after read(A), then this schedule is not recoverable:
  - possibly T_8 finally aborts, and then T_9 has read / processed an *inconsistent* database state

Example:

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Recoverability

Cascading rollback:

- a single transaction failure (i.e. rollback) leads to a series of transaction rollbacks

- Example:
  – if none of the transactions has (yet) committed, then this schedule is (still) recoverable

  $\Rightarrow$ data are consistent

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
abort		read(A)

Recoverability

- A schedule is cascadeless (or avoiding cascading rollback):
  - if cascading rollbacks cannot occur

- That is:
  - for each pair of transactions T_i and T_j, if T_j reads T_i, then the commit operation of T_i appears before the read operation of T_j

- Note the small difference from recoverable schedules !
  - recoverable: the commit operation of T_i appears before the commit operation of T_j

Examples

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)
	commit
commit	

not recoverable
not cascadeless

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)
commit	
	commit

recoverable
not cascadeless

T_1	T_2
read(x)	
write(x)	
commit	
	read(x)
	write(x)
	commit

recoverable
cascadeless

We can prove: cascadeless $\Longrightarrow$ recoverable

The converse is not always true

Concurrency control

- Schedules must be:
  - serializable and recoverable
    (for the sake of database consistency)
  - preferably also cascadeless
    (for the sake of efficiency)
- Conservative (pessimistic) methods:
  - delay (or restart) conflicting transactions
- Locking: make conflicting transactions wait
- Timestamping:
  - roll back and restart conflicting transactions

The timestamping method

- Timestamp: a unique identifier $TS(T_i)$ that indicates the relative starting time of a transaction T_i (created by the DBMS)

- Usually a timestamp is generated:
  - by the system clock, or
  - by incrementing a logical counter for every new transaction

- The main idea of the protocol:
  - timestamps determine the serializability order
  - two transactions T_1, T_2 obtain timestamps $\mathrm{TS}(T_1) < \mathrm{TS}(T_2)$

  $\Longrightarrow$ in the serialized schedule, T_1 is scheduled before T_2

The timestamping method

- The protocol maintains for each data item x two timestamp values:
  - WTS(x): the “Writing Time Stamp”
    - the largest timestamp of a transaction that executed write(x) successfully
  - RTS(x): the “Reading Time Stamp”
    - the largest timestamp of a transaction that executed read(x) successfully
- The protocol ensures that conflicting read and write operations are executed in the timestamp order

The timestamping method

Suppose a transaction *T* issues a read(x):

- if TS(*T*) < WTS(x):
  - an earlier transaction (i.e. *T*) is trying to read a value of an item that has been updated by a later transaction
  - then, the read(x) operation rejects, and *T* is rolled back
- if TS(*T*) ≥ WTS(x):
  - the read(x) operation is executed
  - RTS(x) is set to the maximum of TS(*T*) and the current value for RTS(x)

The timestamping method

Suppose a transaction *T* issues a write(x):

- if TS(*T*) < RTS(x):
  - a later transaction than *T* uses the current value of x and it would be dangerous for *T* to update it
  - then, the write(x) operation rejects, and *T* is rolled back
- if TS(*T*) < WTS(x):
  - transaction *T* attempts to write an obsolete value of x
  - then, the write(x) operation rejects, and *T* is rolled back
- Otherwise:
  - the write(x) operation is executed
  - set WTS(x) = TS(*T*)

Example use of the protocol

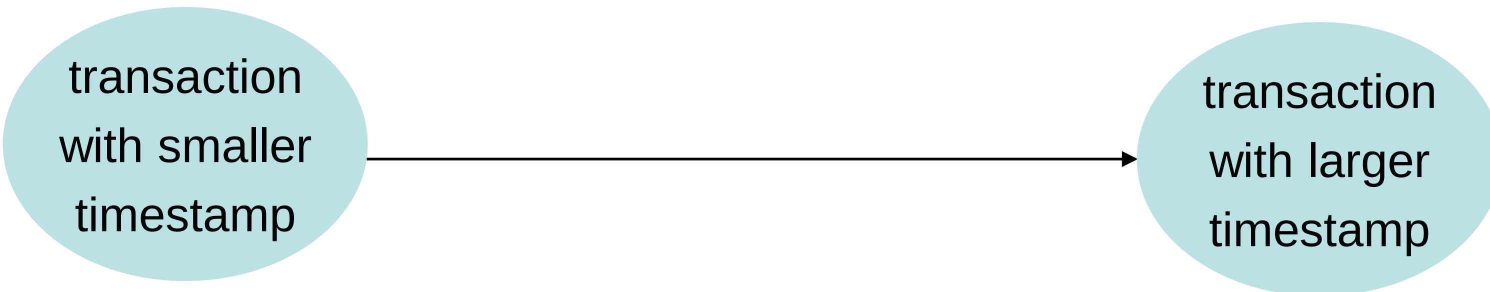
Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
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	

reject / roll back

reject / roll back

Correctness of the protocol

- The timestamping protocol guarantees serializability, since all the arcs in the precedence graph are:



$\Rightarrow$ there is no cycle in the precedence graph

- The timestamping method:
  - ensures freedom from deadlock (as no transaction ever waits)
  - but the schedule may be not recoverable (thus also not cascadeless)

Recoverability / cascade freedom

- Problems with the timestamp protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable
  - Any transaction that reads an item written by T_i must reject
  - This can lead to cascading rollback, i.e. a chain of rollbacks
- Solution:
  - A transaction is structured such that all its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp

Thomas' write rule

- Recall: timestamps guarantee (conflict) serializability (by restarting conflicting operations)
- Can we do better ?
  - i.e. obtain the same results with more concurrency ?
- Yes: by relaxing conflict serializability !
  - instead, we ask for view serializability
- Main idea: the “ignore obsolete write” rule (or “Thomas' write” rule)
- Recall: a non-serial schedule is view serializable if:
  - it is view equivalent (i.e. produces the same results) with a serial schedule

Thomas' write rule

The "ignore obsolete write" rule:

- when a transaction *T* issues a read(x):
  - same as before (in timestamp protocol)

- when a transaction *T* issues a write(x):
  - if TS(*T*) < RTS(x): same as before (*T* is rolled back)
  - if TS(*T*) < WTS(x): then <u>ignore write(x)</u>
  - otherwise: same as before (execute write(x))

- Explanation:
  - a later transaction already updated the value of x

$\Longrightarrow$ *T* attempts to write an out-dated value of x
which <u>will never need to be read</u>

Thomas' write rule

- It can be proved that:
  - Thomas' write rule produces view serializable schedules
  - which can never be produced by the previous protocols

  $\Longrightarrow$ more concurrency $\Longrightarrow$ greater efficiency !

- Idea for correctness:
  - write(x) restarts only if it is read by a later transaction (this would cause false future computations)
  - otherwise write(*x*) is just obsolete – nobody will read it !

- Explanation:
  - a later transaction already updated the value of x

  $\Longrightarrow$ *T* attempts to write an out-dated value of x which <u>will never need to be read</u>

Thomas' write rule: Examples

- Timestamp protocol: T_{11} is rejected and restarted after T_{12}
- Thomas' write rule: write(bal_x) of T_{11} is ignored
  - bal_x is anyway overwritten by T_{12} (and also by T_{13})

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

- view serializable schedule
  - but not conflict serializable

Thomas' write rule: Examples

- In both protocols: T_{20} is rejected and restarted after T_{21}

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	

reject / roll back

Thomas' write rule: Examples

- **Timestamp protocol:** T_{19} is rejected and restarted after T_{21}
- **Thomas' write rule:** write(bal_z) of T_{19} is ignored
  - bal_z is anyway overwritten by T_{21} (as $TS(T_{19}) < TS(T_{21})$)

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}	[illegible]	commit	read(bal_y)	
t_{16}	…20		$bal_y = bal_y + 20$	
t_{17}	[illegible]		write(bal_y)	
t_{18}			commit	

reject / roll back

~~reject / roll back~~

Multiversion Timestamps

- Another generalization of the timestamp protocol:
  - many transactions can simultaneously access an item x
  - but each of them working on a different version of x
  - versions are labeled by timestamps

$\Rightarrow$ increased concurrency

- An operation write(x):
  - either creates a new version of x
  - or aborts and restarts
- An operation read(x):
  - selects the appropriate version of x
  - always successful

Multiversion Timestamps

- A data item x has a sequence of versions $\langle x_1, x_2, \ldots, x_m \rangle$
- Each version x_k contains:
  - Content: the value of x_k
  - WTS(x_k): "write-timestamp"
    - the timestamp of the transaction that created version x_k
  - RTS(x_k): "read-timestamp"
    - the largest timestamp of a transaction that has read version x_k
- When transaction *T* creates a new version x_k of x, WTS(x_k) and RTS(x_k) are initialized to TS(*T*)
- RTS(x_k) is updated whenever:
  - a transaction *T* reads x_k and TS(*T*) > RTS(x_k)

Multiversion Timestamps

- Consider a transaction *T*
- Let x_k be the version of x where:
  - $WTS(x_k)$ is the largest write timestamp that is $\leq TS(T)$
- Transaction *T* issues a **read**(*x*):
  - the value of version x_k is returned

    (always successful)

Multiversion Timestamps

- Consider a transaction T

- Let x_k be the version of x where:
  - $WTS(x_k)$ is the largest write timestamp that is $\leq TS(T)$

- Transaction T issues a **write**(x):
  - if $TS(T) < RTS(x_k)$, then transaction T is rolled back

    (otherwise the transaction that has last read version x_k, will never see the update of T $\Longrightarrow$ no serializability)

  - if $TS(T) \geq RTS(x_k)$ then:
    - if $TS(T) = WTS(x_k)$, the contents of x_k are overwritten

      (x_k was created previously by the same transaction T)

    - if $TS(T) > WTS(x_k)$, a new version of x is created

Multiversion Timestamps

- Versions can be deleted if no longer required

- To determine whether a version of data item x is required:
  - find the timestamp $TS(T)$ of the oldest alive transaction
  - for any two versions x_i, x_j of x:
    - if WTS(x_i) and WTS(x_j) are both $< TS(T)$
      then delete the oldest of x_i, x_j (i.e. with the oldest WTS)

- Main idea:
  - keep only one version of x that is older than $TS(T)$,
    just for the case T needs to rollback

Summary of the Lecture

- Concurrency control methods:
  - recoverable schedules
  - casrading rollback
    - cascedeless schedules
  - the timestamping protocol
  - Thomas' write rule
  - the multiversion timestamp ordering