Concurrency control in Homogeneous Distributed Databases (2)

- Timestamp ordering
- Basic implementation
- Optimistic CC in distributed DB
- Distributed deadlock detection

based on slides by Weikum / Vossen: Transactional Information Systems; H. Garcia Molina

---

Non-locking concurreny control

Time stamp ordering

Basic idea:

- assign timestamp when transaction starts
- if ts(t1) < ts(t2) … < ts(tn), then scheduler has to produce history equivalent to t1,t2,t3,t4, ... tn

Timestamp ordering rule:

If pi[x] and qj[x] are conflicting operations,
then pi[x] is executed before qj[x] (pi[x] < qj[x])
iff ts(ti) < ts(tj)

Issue: how to find out that x has been modified by a younger / older TA ?? Timestamp for each data item !

Timestamp ordering

- TO concurrency control guarantees conflict-serializable schedules:

If not: cycle in conflict graph
cycle of length 2: ts(t1) < ts(t2) $\wedge$ ts(t2) < ts(t1) #

induction over length of cycle => #

=> No cycle in conflict graph ✓

Example: Distributed case

(Node S1)	(Node S2)
(t1) a ← X(S1)	(t2) d ← Y(S2)
(t1) X ← a+100	(t2) Y ← 2d
(t2) c ← X(S1)	(t1) ~~b ← Y(S2)~~
(t2) X ← 2*c	(t1) ~~Y ← b+100~~

read: t1 reads X@S1 into a ... etc

ts(t1) < ts(t2)

Abort t1 at S1 ← Abort t1

Cascading abort of t2 → Abort t2 at S2

Strict timestamp ordering

- Strict TO

Lock the items changed until ta has bee committed (or aborted)

(Node S1)	(Node S2)
(t1) a ← X(S1)	(t2) d ← Y(S2)
(t1) X ← a+100	(t2) Y ← 2d
LOCK X	(t1) ~~b ← Y(S2)~~
abort t1 at S1	(t1) ~~Y ← b+100~~

abort t1 at S1 ← ts(t1) < ts(t2) abort t1

UNLOCK T1
(t2) c ← X(S1)
(t2) X ← 2*c

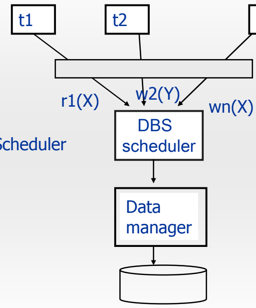
TO Scheduler

- Basic principle:

Abort transaction if its operation is "too late“
Remember timestamp of last write of X: maxW[X]
and last read maxR[X]

Transaction i: ti with timestamp ts(ti)
Operations: ri(X) / wi(X) - ti wants to read / write X
Scheduler state: maxR[X] / maxW[]
timestamp of youngest TA which read X / has written X

Scheduler / Data manager architecture



Global TA manager

Local Scheduler

Independent processes / threads

- scheduler: ensure serializable sequence of operations
- data manager: record access, buffer management, I/O

TO Scheduler: read

Read: TA ti with timestamp ts(ti) reads X : ri(X)

maxW[X] > ts(ti):
there is a younger TA wich has written X

⇨ contradicts timestamp ordering:
ti reads too late

⇨ abort TA ti , restart ti

maxW[x] < ts(ti) ⇨ go ahead ?

Example: ------|------|----------- >

ri(X) wj(X) ts(ti) < ts(tj)

but data manager may execute wj(X) before ri(X)

TO Scheduler: read

Write: TA ti with timestamp ts(ti) writes X : wi(X)

maxW[X] > ts(ti) $\vee$ maxR[X] > ts(ti) :
/* but X has been *written or read by younger transaction*:

⇨ # timestamp ordering
⇨ abort TA ti

otherwise: ⇨ schedule wi(X) for execution

- Same issue as with 'read'
- Solution: 'lock variables"
  number of Readers of X : nR[X]
  number of Writers of X: nW[X]

nR[X] == 0 $\wedge$ nW[X] == 0 : schedule wi(X)
otherwise enqueue wi(X)

similar for reads…

TO scheduling

Note:

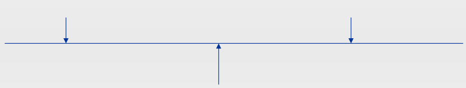
- This kind of blocking reads / writes not necessary with 2PL since 2PL lock manager blocks anyway
- Different from waiting for EOT to enforce strict TO protocol

  Can be implemented with nW[X]: if ti has incremented nW[X] 0 → 1
  wait for commit (ti) before decrementing nW[X]
- Enqueue operations ri(X) / wi(X)
  if nW[X] > 0 / nW[X] > 0 $\vee$ nR[X] > 0
- Release appropriate element from queue if nW[X] and/or nR[X] decreased

Thomas Write Rule

- Idea: younger write overwrites older write without changing effect of timestamp ordering

maxR[X] maxW[X]



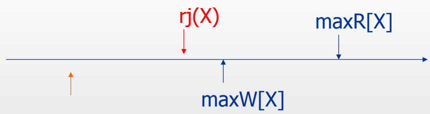
ts(T_i)

t_i wants to write X

Modified Time ordered scheduling of writes

```
Queue q; ….
void write ( TA ti, Object X) {   /* write wi(X)
  if ts(ti)< maxR[X]   ti.abort();
  else  if ts(Ti)<maxW[X] ;
        /* IGNORE WRITE (tell ti "success")
        else {    /* process write as before…
        maxW[X] ← ts(ti);
        if (q.isEmpty() ∧ nW[x]== 0 ∧ nR[X] ==0 ) {
          nW[X] ← 1;    dataMgr.write(X);
          WAIT ( commit (ti)) ;
         }
       else q.add (W, ti) ;
       }
 }
```

Another rule?



ts(ti)
TA ti with ts(ti) < maxW[X] < maxR[X] wants to write

Can ti go ahead and ignore wi(X) ??

If there is a read rj(X) before the write → maxW[X]
then TA tj has read a wrong value:
wrong order if tj < ti

Management of timestamps

data	maxR	maxW	nR / nW	
X1				
X2				
⋮	⋮	⋮		
Xn				

maxR / maxW must be kept for each data item in DB
Space! I/O traffic !!

Timestamp cache

data	maxR	maxW	nR / nW	
X1				
X2				
⋮	⋮	⋮		
Xn				

tsMin

- if TA reads / writes data item: make cache entry
- flush cache periodically: purge rows with timestamp <= tsMin e.g. currentTime – delta
- ts for data on disk: tsMin

Timestamp cache

- Enforce timestamp order rule for wi(X) :
  - Use maxW (X) if X is in cache
  - Assume maxW(X) = tsMin otherwise
  - Same for reads
- Hash table as data structure (like lock table)

Time stamp order and distribution

- Distributed TO scheduling
  - Basic prerequisite: *total order of TA timestamps*
    e.g. local counter ++ server#
  - TO schedulers independent
  - Total order guarantees serializable read / write at all sites
  - At TA commit: release nW(X) / nR(X) locks
  - Tricky detail:
    suppose only a few TA at site S1, many at site S2
    ⇨ counter[S1] << counter[S2]
    ⇨ TAs originating at S2 will be frequently aborted
  - Any idea to solve this problem?

Time stamp cc and 2PL

TO ⇨ conflict serializable
2PL ⇨ conflict serializable

2PL = TO ??
NO!

t1: w1[Y]
t2: r2[X] r2[Y] w2[Z]   ts(t1)<ts(t2)<ts(t3)
t3: w3[X]

S: r2[X] w3[X] w1[Y] r2[Y] w2[Z]

S could be produced with T.O. but not with 2PL

Pessimistic vs Optimistic

- Timestamp Order is pessimistic
  - All checks are made *before* operation is scheduled
  - Optimistic: work isolated on copy of data
    write back if no potential conflict
    has occurred

Optimistic protocol

- Optimistic CC in homogeneous DDBS
- Same protocol as centralized DBS
  - Read phase
  - Validation phase:
    ReadSet (tj | tj running) $\cap$ WriteSet (validation TA) == $\emptyset$?
    Backward oriented optimistic CC (BOCC)
- Issue: Validation of TAs on different Servers in the same order

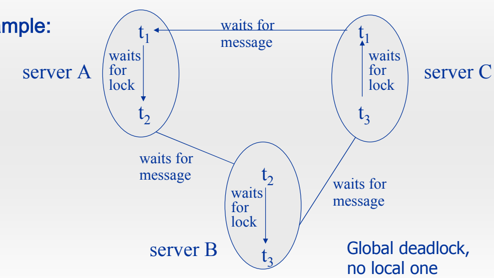
CC in homogeneous DDBS

- 2PL
  - Used frequently in commercial systems
  - Simple enhancement in distributed systems
  - Deadlocks possible – critical in distributed DBS
- Timestamp ordering
  - a reasonable alternative
  - aborts more likely -
  - no deadlocks -
  - useful in a distributed sytem
- Optimistic
  - become popular in widely distributed systems

Distributed Deadlocks

Deadlock detection in DDB

Example:



Global deadlock, no local one

Expl by Weikum / Vossen

Distributed Deadlocks

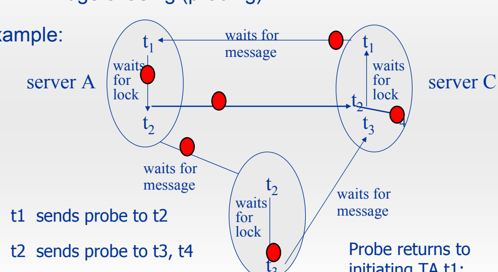
A. Timeout
  - used in most commercial systems
  - Abort transaction if TA waits longer than specified timeout. Victim?

B. Centralized DL detection
  - one site S is responsible for DL detection
  - other sites send periodically local Wait-For-graphs to S
  - S forms Global WF-graph and checks for cycles

Distibuted Deadlock detection

C. Distributed detection
  - Edge chasing (probing)

Example:



t1 sends probe to t2

t2 sends probe to t3, t4

t3 sends probe to t1

Probe returns to initiating TA t1: deadlock!

Distributed deadlock detection

- Path pushing

Algorithm:

- each node that has a wait-for path from transaction t_i to t_j such that t_i has an *incoming wait-for-message edge* and t_j has an *outgoing wait-for-message edge*, sends the path to the server along the outgoing edge (if the id (or timestamp) of t_i is smaller than the id of t_j)
- upon *receiving a path*, each node *concatenates it with its local paths* and forwards it along outgoing edges
- if there is a cycle among n servers, *at least one server will detect the cycle* after at most n such rounds

2PL: no false deadlocks

Distributed deadlock detection: path pushing

- Example from above

server A $\xrightarrow{t_1 \to t_2}$ server B $\xrightarrow{t_2 \to t_3}$ server C

server B $\xrightarrow{t_1 \to t_2 \to t_3}$ server C

knows $t_3 \to t_1$ locally and detects global deadlock

Summary

- Homogenous concurrency control
  - Slight extension of centralized protocols
  - Always possible to introduce some kind of centralized control
  - Contradicts principles of avoiding single point of failure and scalability
  - TS ordering with little overhead
  - Deadlock detection: expensive or time-out