Concurrency control in Homogeneous Distributed Databases (2)

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

Non-locking concurrency 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(pi) < ts(qj)

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

Example: Distributed case

(Node S1)
(t1) a ← X(S1)
(t1) X ← a + 100
LOCK X
abort t1 at S1
unlock T1

(Node S2)
(t2) d ← Y(S2)
(t2) Y ← 2d

read: t1 reads X(S1) into a ... etc

Abort t1 at S1
Abort t2 at S2
Cascading abort of t2

Strict timestamp ordering

Strict TO
Lock the items changed until ta has been committed (or aborted)

(Node S1)
(t1) a ← X(S1)
(t1) X ← a + 100
LOCK X
abort t1 at S1

(Node S2)
(t2) d ← Y(S2)
(t2) Y ← 2d

Scheduler state: maxR[X] / maxW[X]

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

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

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]
**Scheduler / Data manager architecture**

Local Scheduler

DBS scheduler

Data manager

Global TA manager

**TO Scheduler: read**

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

maxW[X] > ts(ti) ∨ 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 ∧ nW[X] == 0: schedule wi(X)

otherwise enqueue wi(X)

similar for reads…

**Thomas Write Rule**

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

maxR[X] maxW[X]  

```
  ts(Ti)  

  ti 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?

\[ \eta(X) \rightarrow \text{maxR}[X] \rightarrow \text{maxW}[X] \rightarrow \text{ts}(ti) \]

TA \( ti \) with \( ts(ti) < \text{maxW}[X] < \text{maxR}[X] \) wants to write

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

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

Management of timestamps

<table>
<thead>
<tr>
<th>data</th>
<th>maxR</th>
<th>maxW</th>
<th>nR / nW</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Timestamp cache

<table>
<thead>
<tr>
<th>data</th>
<th>maxR</th>
<th>maxW</th>
<th>nR / nW</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \text{tsMin} \)

- if TA reads / writes data item: make cache entry
- flush cache periodically: purge rows with
  timestamp \( \leq \text{tsMin} \) e.g. currentTime - delta
- ts for data on disk: \( \text{tsMin} \)

Timestamp cache

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

Timestamp order and distribution

- Distributed TO scheduling
  - Basic prerequisite: total order of TA timestamps
  - 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
  - \( \text{counter}[S1] < \text{counter}[S2] \)
  - TAs originating at S2 will be frequently aborted
- Any idea to solve this problem?

Time stamp cc and 2PL

\( \text{TO} \Rightarrow \text{conflict serializable} \)
\( 2PL \Rightarrow \text{conflict serializable} \)

\( 2PL = \text{TO} ?? \)
NO!

\( t1: \ w1[Y] \)
\( t2: \ r2[X] \ r2[Y] \ w2[Z] \hspace{1cm} \text{ts}(t1) < \text{ts}(t2) < \text{ts}(t3) \)
\( t3: \ w3[X] \)

\( S: \ r2[X] \ w3[X] \ w1[Y] \ r2[Y] \ w2[Z] \)

\( S \) 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: $\text{ReadSet}(t_j | t_j \text{ running}) \cap \text{WriteSet}(\text{validation TA}) = \emptyset$?
  - Backward oriented optimistic CC (BOCC)
- Issue: Validation of TAs on different Servers in the same order

**Optimistic protocol**

- Optimistic CC in homogeneous DDBS
- Same protocol as centralized DBS
  - Read phase
  - Validation phase: $\text{ReadSet}(t_j | t_j \text{ running}) \cap \text{WriteSet}(\text{validation TA}) = \emptyset$?
  - 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 system
- Optimistic
  - become popular in widely distributed systems

**Distributed Deadlocks**

**Deadlock detection in DDB**

**Example:**

- **Server A**
  - $t_1$ waits for lock
  - $t_2$ waits for message
- **Server B**
  - $t_2$ waits for lock
  - $t_3$ waits for message
- **Server C**
  - $t_1$ waits for lock
  - $t_3$ waits for message

Global deadlock, no local one

**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

**Distributed Deadlock detection**

**C. Distributed detection**
- Edge chasing (probing)

**Example:**

- Server A
  - $t_1$ waits for lock
  - $t_4$ waits for message
- Server B
  - $t_3$ waits for lock
  - $t_4$ waits for message
- Server C
  - $t_1$ waits for lock
  - $t_3$ waits for message

$\text{t}_1$ sends probe to $\text{t}_2$, $\text{t}_3$ sends probe to $\text{t}_1$, $\text{t}_4$ sends probe to $\text{t}_3$

Probe returns to initiating TA $\text{t}_1$: 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.

Example from above

server A

\[ t_1 \rightarrow t_2 \]

server B

\[ t_2 \rightarrow t_3 \]

server C

\[ t_1 \rightarrow t_3 \rightarrow t_1 \]

knows \(t_1 \rightarrow t_3\) 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