Who cares about data consistency?
Who cares about data consistency?

POST-MORTEM OF O DURABILITY ISSUE ON AUGUST 6 & 7, 2013
Discussion in 'News & Announcements' started by Phobos, Aug 8, 2013.
Overview

• Concurrency Control
  – Schedules and Serializability
  – Locking
  – Timestamp Control

• Failure Recovery
  – next set of notes...
Schedule

• Just like in a centralized system, a schedule represents how a set of transactions were executed
• Schedules may be “good” or “bad” (preserve constraints)
Example

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th></th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(T_1) a \leftarrow X$</td>
<td>5</td>
<td>$(T_2) c \leftarrow X$</td>
</tr>
<tr>
<td>2</td>
<td>$(T_1) X \leftarrow a+100$</td>
<td>6</td>
<td>$(T_2) X \leftarrow 2c$</td>
</tr>
<tr>
<td>3</td>
<td>$(T_1) b \leftarrow Y$</td>
<td>7</td>
<td>$(T_2) d \leftarrow Y$</td>
</tr>
<tr>
<td>4</td>
<td>$(T_1) Y \leftarrow b+100$</td>
<td>8</td>
<td>$(T_2) Y \leftarrow 2d$</td>
</tr>
</tbody>
</table>

constraint: $X=Y$

Precedence relation
Schedule S1

(node X)

1 (T₁) a ← X
2 (T₁) X ← a+100
5 (T₂) c ← X
6 (T₂) X ← 2c

(node Y)

3 (T₁) b ← Y
4 (T₁) Y ← b+100
7 (T₂) d ← Y
8 (T₂) Y ← 2d

If X=Y=0 initially, X=Y=200 at end (always good?)
Definition of Schedule

Let $T = \{T_1, T_2, T_n\}$ be a set of transactions. A schedule $S$ over $T$ is a partial order with ordering relation $<_S$ where:

1. $S = \bigcup_{i=1}^{N} T_i$
2. $<_S \subseteq \bigcup_{i=1}^{N} <_i$
3. for any two conflicting operations $p, q \in S$, either $p <_S q$ or $q <_S p$
Example

\((T_1)\) \(r_1[X] \rightarrow W_1[X]\)

\((T_2)\) \(r_2[X] \rightarrow W_2[Y] \rightarrow W_2[X]\)

\((T_3)\) \(r_3[X] \rightarrow W_3[X] \rightarrow W_3[Y] \rightarrow W_3[Z]\)

\(S_1:\) \(r_3[Y] \rightarrow W_3[X] \rightarrow W_3[Y] \rightarrow W_3[Z]\)

\(r_1[X] \rightarrow W_1[X]\)
Definition of \( P(S) \)

- The precedence graph for schedule \( S, P(S) \), is a directed graph where
  - nodes: the transactions in \( S \)
  - edges: \( T_i \rightarrow T_j \) is an edge IFF
    \( \exists p \in T_i, q \in T_j \) such that
    \( p, q \) conflict and \( p <_S q \)
Example

\[ S_1: \quad r_1[X] \rightarrow w_1[X] \rightarrow w_1[Y] \]
\[ r_2[X] \rightarrow w_2[Y] \]
\[ r_3[X] \rightarrow w_3[X] \]

\[ P(S_1): \quad T_2 \rightarrow T_1 \rightarrow T_3 \]
Serializability Theorem

Theorem: A schedule S is serializable IFF $P(S)$ is acyclic.
Enforcing Serializability

- Locking
- Timestamps
Locking

- Just like in a centralized system...
- But with multiple lock managers

node 1

node 2

D₁

D₂

D₁

D₂

scheduler 1

scheduler 2

locks

locks

for

for

D₁

D₂
Locking

- Just like in a centralized system...
- But with multiple lock managers

![Diagram showing locking mechanism with multiple nodes and lock managers.](image-url)
Locking Rules

• Well-formed transactions
• Legal schedulers
• Two-phase transactions

• These rules guarantee serializable schedules
What about

• Locking in a shared-memory architecture?
• Locking in a shared-disks architecture?
Locking with Shared Memory

- Where does lock table live?
- How do we avoid race conditions?
Locking with Shared Disks

- Where does lock table live?
- How do we avoid race conditions?
Shared Disk Locking, Cases to Discuss

• Control of Data Partitioned; fixed partition function known by everyone

• Dynamic partition of control
  – For each DB object i we need
    • LT(i): lock table entry for i (transaction that has lock, waiting transactions, lock mode, etc...)
    • W(i): at what processor is LT(i) currently?
  – Replicate W(i) at all processors (why?)
  – Need replicated data management scheme!
Overview

• Concurrency Control
  – Schedules and Serializability
  – Locking
  – Timestamp Control

• Failure Recovery
  – next set of notes...
Timestamp Ordering Schedulers

- **Basic idea:**
  - assign timestamp as transaction begins
  - if \( ts(T_1) < ts(T_2) \ldots < ts(T_n) \), then scheduler produces history equivalent to \( T_1, T_2, T_3, T_4, \ldots T_n \)
TO Rule

If \( p_i[x] \) and \( q_j[x] \) are conflicting operations, then \( p_i[x] \) is executed before \( q_j[x] \)

\[ (p_i[x] <_S q_j[x]) \]

IFF \( ts(T_i) < ts(T_j) \)
Example: a non-serializable schedule $S_2$

(Node X)

- $(T_1)$ $a \leftarrow X$
- $(T_1)$ $X \leftarrow a + 100$
- $(T_2)$ $c \leftarrow X$
- $(T_2)$ $X \leftarrow 2c$

(Node Y)

- $(T_2)$ $d \leftarrow Y$
- $(T_2)$ $Y \leftarrow 2d$
- $(T_1)$ $b \leftarrow Y$
- $(T_1)$ $Y \leftarrow b + 100$
Example: a non-serializable schedule $S_2$

$$ts(T_1) < ts(T_2)$$

(Node X)

- $(T_1)\ a \leftarrow X$
- $(T_1)\ X \leftarrow a+100$
- $(T_2)\ c \leftarrow X$
- $(T_2)\ X \leftarrow 2c$

(Node Y)

- $(T_2)\ d \leftarrow Y$
- $(T_2)\ Y \leftarrow 2d$
- $(T_1)\ b \leftarrow Y$
- $(T_1)\ Y \leftarrow b+100$

reject!

abort $T_1$
Example: a non-serializable schedule $S_2$

$$ts(T_1) < ts(T_2)$$

<table>
<thead>
<tr>
<th>(Node X)</th>
<th>(Node Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_1) $a \leftarrow X$</td>
<td>(T_2) $d \leftarrow Y$</td>
</tr>
<tr>
<td>(T_1) $X \leftarrow a+100$</td>
<td>(T_2) $Y \leftarrow 2d$</td>
</tr>
<tr>
<td>(T_2) $c \leftarrow X$</td>
<td>(T_1) $b \leftarrow Y$</td>
</tr>
<tr>
<td>(T_2) $X \leftarrow 2c$</td>
<td>(T_1) $Y \leftarrow b+100$</td>
</tr>
</tbody>
</table>

 abort $T_2$  
 abort $T_1$  
 abort $T_1$  
 abort $T_2$  
 reject!
Strict T.O.

- Lock written items until it is certain that writing transaction has been successful (avoid cascading rollbacks)
Example Revisited

\[
\begin{align*}
(T_1) & \quad a \leftarrow X \\
(T_1) & \quad X \leftarrow a+100 \\
(T_2) & \quad c \leftarrow X \quad \text{delay}
\end{align*}
\]

\[
\begin{align*}
(T_2) & \quad d \leftarrow Y \\
(T_2) & \quad Y \leftarrow 2d \\
(T_1) & \quad b \leftarrow Y \quad \text{reject!} \\
& \quad \text{abort } T_1
\end{align*}
\]

\[ts(T_1) < ts(T_2)\]
Example Revisited

\[
\begin{align*}
\text{(Node X)} & \\
(T_1) & a \leftarrow X \\
(T_1) & X \leftarrow a + 100 \\
(T_2) & c \leftarrow X \quad \text{delay} \\
\text{(Node Y)} & \\
(T_2) & c \leftarrow X \\
(T_2) & X \leftarrow 2c
\end{align*}
\]

\[\text{ts}(T_1) < \text{ts}(T_2)\]

\[
\begin{align*}
\text{(Node Y)} & \\
(T_2) & d \leftarrow Y \\
(T_2) & Y \leftarrow 2d \\
(T_1) & b \leftarrow Y \quad \text{reject!} \\
\text{aborted} & T_1
\end{align*}
\]
Enforcing T.O.

• For each data item X:

\[
\begin{align*}
\text{MAX}_R[X]: & \text{ maximum timestamp of a transaction that read } X \\
\text{MAX}_W[X]: & \text{ maximum timestamp of a transaction that wrote } X \\
\text{rL}[X]: & \text{ # of transactions currently reading } X (0,1,2,...) \\
\text{wL}[X]: & \text{ # of transactions currently writing } X (0 \text{ or } 1)
\end{align*}
\]
T.O. Scheduler - Part 1

ri[X] arrives
  IF ts(Ti) < MAX_W[X] THEN ABORT Ti
  ELSE [ IF ts(Ti) > MAX_R[X] THEN MAX_R[X] ← ts(Ti);
      IF queue is empty AND wL[X] = 0 THEN
        [rL[X] ← rL[X]+1; START READ OF X]
      ELSE add (r, Ti) to queue]
T.O. Scheduler - Part 2

Wi[X] arrives

IF ts(Ti) < MAX_W[X] OR ts(Ti) < MAX_R[X]
THEN ABORT Ti

ELSE [ MAX_W[X] ← ts(Ti);
IF queue is empty AND wL[X]=0 AND rL[X]=0
THEN [wL[X] ← 1; WRITE X;
    WAIT FOR Ti TO FINISH ]
ELSE add (w, Ti) to queue ]
T.O. Scheduler - Part 3

When o finishes (o is r or w) on X

\[ oL[X] \leftarrow oL[X] - 1; \ NDONE \leftarrow \ TRUE \]

WHILE NDONE DO

[ let head of queue be \((q, T_j)\); \ (smallest timestamp) ]

IF \(q=w\) AND \(rL[X]=0\) AND \(wL[X]=0\) THEN

[remove \((q, T_j)\); \(wL[X] \leftarrow 1;\)

WRITE X AND WAIT FOR \(T_j\) TO FINISH ]

ELSE IF \(q=r\) AND \(wL[X]=0\) THEN

[remove \((q, T_j)\); \(rL[X] \leftarrow rL[X] +1;\) START READ OF X]

ELSE NDONE \leftarrow FALSE ]
Note about the code

for reads:  [rL[X] ← rL[X]+1;  START READ OF X]

for writes:  [wL[X] ← 1;  WRITE X;
               WAIT FOR Ti TO FINISH ]

Meaning: In Part 3, the end of a write is only processed when all writes for its transaction have completed.
If a transaction is aborted, it must be retried with a new, larger timestamp.

\[
\begin{align*}
\text{MAX}_R[X] &= 10 \\
\text{MAX}_W[X] &= 9 \\
\text{read } X \\
\text{ts}(T) &= 8
\end{align*}
\]
If a transaction is aborted, it must be retired with a **new, larger** timestamp.

- $\text{MAX}_R[X] = 10$
- $\text{MAX}_W[X] = 9$
- $X$ read
- $\text{ts}(T) = 8$
- $\text{ts}(T) = 11$

Starvation possible
Thomas Write Rule

MAX_R[X]  MAX_W[X]

ts(T_i)

T_i wants to write X
Change in T.O. Scheduler:

When Wi[X] arrives

IF ts(Ti)<MAX_R[X] THEN ABORT Ti
ELSE IF ts(Ti)<MAX_W[X] THEN
    [IGNORE* THIS WRITE (tell Ti it was OK)]
ELSE [process write as before…
    MAX_W[X] ← ts(Ti);
    IF queue is empty AND wL[X]=0 AND rL[X] =0 THEN
        [wL[X] ← 1; WRITE X and WAIT FOR Ti TO FINISH]
ELSE add (W, Ti) to queue]

* Ignore if transaction that wrote MAX_X[X] did not abort
Question

- Why can’t we let Ti go ahead?
Question

Ti wants to write

MAX_W[X]

MAX_R[X]

Tj reads

ts(Ti)

Why can’t we let Ti go ahead?

MAX_R[X] is only the latest read; there could be a Tj read as shown...
Optimizations

- Update MAX_R, MAX_W when action executed, not when action put on queue

Example: \[ \text{MAX}_W[X]=9 \text{ or } 7? \]

\[
\begin{array}{c|c}
X: & W, ts=9 \\
    & W, ts=8 \\
    & W, ts=7 \\
\end{array}
\]

↑

active write
Optimizations

- Use multiple versions of data

<table>
<thead>
<tr>
<th>X:</th>
<th>Value written with ts=9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value written with ts=7</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>r_i[x]</td>
<td>ts(T_i)=8</td>
</tr>
</tbody>
</table>

Value written with ts=7
2PL ≠ TO

T₁: w₁[Y]
T₂: r₂[X] r₂[Y] w₂[Z] \( \text{ts}(T₁)<\text{ts}(T₂)<\text{ts}(T₃) \)
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
Are all 2PL schedules T.O.?

any examples here??

previous example
Theorem

If $S$ is a schedule representing an execution by a T.O. scheduler, then $S$ is serializable

Proof:

1) say $T_i \rightarrow T_j$ in $P(S)$

$\Rightarrow \exists$ conflicting $p_i[x], q_j[x]$ in $S$,

such that $p_i[x] <_{S} q_j[x]$

Then by T.O. rule, $ts(T_i) < ts(T_j)$
Proof - Continued

2) Say there is a cycle
   \[ T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \] in \( P(S) \)
   then:
   \[ ts(T_1) < ts(T_2) < ts(T_3) \ldots < ts(T_n) < ts(T_1) \]
   A contradiction!

3) So \( P(S) \) is acyclic
   \[ \Rightarrow S \text{ is serializable} \]
# Timestamp management

<table>
<thead>
<tr>
<th>Item</th>
<th>data</th>
<th>MAX_R</th>
<th>MAX_W</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>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Xn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- too much space!
- more IO
### Timestamp cache

<table>
<thead>
<tr>
<th>Item</th>
<th>MAX_R</th>
<th>MAX_W</th>
<th>tsMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If transaction reads or writes X, make entry in cache for X (add row if not in)
- Periodically purge all items X with MAX_R[X] < tsMIN, MAX_W[X] < tsMIN and remember tsMIN (choose tsMIN ≈ current time - d)
Timestamp cache

- To enforce T.O. rule for pi[X]
  - if X entry in cache,
    use MAX_R, MAX_W values in cache
  - else assume
    MAX_R[X]=tsMIN, MAX_W[X]=tsMIN

- Use hashing (on items) for cache
  (same as lock table)
Each scheduler is “independent”
At end of transaction, signal all schedulers involved to release all wL[X] locks
Distributed T.O. Scheduler

- Each scheduler is “independent”
- At end of transaction, signal all schedulers involved to release all wL[X] locks
Summary

• 2PL  
  - the most popular  
  - useful in a distributed system  
  - deadlocks possible  
  - several variations

• T.O.  
  - good for multiple versions  
  - aborts more likely  
  - no deadlocks  
  - useful in a distributed system
• Others concurrency control schemes
e.g., Certifiers, serialization graph testing

  hard to implement in a distributed system

  not very practical need global data structure