Notes 5: Concurrency Control
Topics

Data ✔
   Database design

Queries ✔
   Decomposition
   Localization
   Optimization

Transactions
   Concurrency control
   Reliability
   Replication
Transactions

*Programs with database accesses*

→ Always end in commit or abort
Transactions

*Programs with database accesses*

→ Always end in commit or abort

**Properties**

- Atomicity
- Consistency
- Isolation
- Durability
Transactions

*Programs with database accesses*

→ Always end in commit or abort

Concurrent control

**Properties**
- Atomicity
- Consistency
- Isolation
- Durability

Reliability
Concurrency Control

Synchronization primitives
Mutual exclusive access
Execution ordering

Pessimistic
Optimistic
Concurrency Control

Schedules and serializability
Locking
Timestamp ordering
Schedule

A schedule represents how a set of transactions are executed

Just like in a centralized system

Schedules may be good (i.e., preserve constraints) or bad
Schedule

Example

\[
\begin{align*}
T_1 & \\
1 & (T_1) a \leftarrow X \\
2 & (T_1) X \leftarrow a + 100 \\
3 & (T_1) b \leftarrow Y \\
4 & (T_1) Y \leftarrow b + 100 \\
T_2 & \\
5 & (T_2) c \leftarrow X \\
6 & (T_2) X \leftarrow 2c \\
7 & (T_2) d \leftarrow Y \\
8 & (T_2) Y \leftarrow 2d
\end{align*}
\]

Node X

Node Y

constraint: X = Y

↓ Precedence relation
Schedule

Schedule $S$

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

If $X = Y = 0$ initially, $X = Y = 200$ at the end

Precedence

↓ Intra-transaction
↓ Inter-transaction
# Schedule

**Schedule S**

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

*If \(X = Y = 0\) initially, \(X = Y = 200\) at the end*

**Precedence**
- ↓ Intra-transaction
- ↓ Inter-transaction

**Other** \(T_1, T_2\) schedules with different outcomes?
Schedule

Formal definition

Let $T = \{ T_1, T_2, \ldots, T_n \}$ 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) $<_S \supseteq \bigcup_{i=1}^{n} <_{T_i}$

3) For any two conflicting operations $p, q \in S$,

   either $p <_S q$ or $q <_S p$

   On same data, at least one is a write
Schedule

Example

\( T_1 \quad r_1[X] \rightarrow w_1[X] \)

\( T_2 \quad r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X] \)

\( T_3 \quad r_3[X] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z] \)

\( r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X] \)

\( r_3[X] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z] \)

\( r_1[X] \rightarrow w_1[X] \)
Schedule

Precedence graph

The precedence graph \( P(S) \) for some schedule is a directed graph

**Nodes**
the transactions in \( S \)

**Edges**
\( T_i \rightarrow T_j \) is an edge iff

\[ \exists p \in T_i, q \in T_j \text{ such that } p, q \text{ conflict and } p <_S q \]
**Schedule**

**Example**

\[ r_3[X] \rightarrow w_3[X] \]

\[ S \quad r_1[X] \rightarrow w_1[X] \rightarrow w_1[Y] \]

\[ r_2[X] \rightarrow w_2[Y] \]

\[ T_2 \rightarrow T_1 \rightarrow T_3 \]

**P(S)**

- \( T_2 \rightarrow T_1 \) because \( r_2[X] \rightarrow w_1[X] \), \( w_2[Y] \rightarrow w_1[Y] \)
- \( T_1 \rightarrow T_3 \) because \( w_1[X] \rightarrow r_3[X] \), \( w_1[X] \rightarrow w_3[X] \)
- \( T_2 \rightarrow T_3 \) because \( r_2[X] \rightarrow w_3[X] \)
Serializability

Theorem
A schedule $S$ is (conflict-)serializable iff $P(S)$ is acyclic
Serializability

Enforcement
Locks
Timestamps
Locking

Just like in a centralized system
But with multiple lock managers

![Diagram of locking in a distributed system with multiple lock managers on different nodes.](image)
Locking

Just like in a centralized system
But with multiple lock managers

scheduler 1
D₁ locks
node 1

scheduler 2
D₂ locks
node 2

access & lock data

T
release all locks at the end
Locking

Reminder
Using locks alone does not guarantee serializability
Locking

Reminder
Using locks alone does not guarantee serializability

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

Node X

1 \( L(X) \) \((T_1)\) \(a \leftarrow X\)

2 \( R(X) \) \((T_1)\) \(X \leftarrow a+100\)

3 \( L(X) \) \((T_2)\) \(c \leftarrow X\)

4 \( R(X) \) \((T_2)\) \(X \leftarrow 2c\)

Node Y

5 \( L(Y) \) \((T_2)\) \(d \leftarrow Y\)

6 \( R(Y) \) \((T_2)\) \(Y \leftarrow 2d\)

7 \( L(Y) \) \((T_1)\) \(b \leftarrow Y\)

8 \( R(Y) \) \((T_1)\) \(Y \leftarrow b+100\)

If \(X = Y = 0\) initially, \(X = 200\) and \(Y = 100\) at the end \(\neq\) serializable
Two-Phase Locking

One solution
Two-Phase Locking

One solution

May lead to cascading aborts
Strict Two-Phase Locking

Only release on commit or abort to avoid cascading aborts
Locking with Shared Memory

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

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

Fixed partition

Dynamic partition
For each data item $X$ need to have lock table entry $L(X)$
For each $L(X)$ need to know current owner processor $P(X)$
$\rightarrow$ Replicate $P(X)$ at all processors
Deadlocks

Remember, 2PL may lead to deadlocks

\[ T_1 \quad L(X), r(X), L(Y) \]
\[ T_2 \quad L(Y), r(Y), L(X) \]

Need to avoid cycles in wait-for graph (WFG) between transactions

Many deadlock solutions
  Detection vs. prevention
    Timeouts
    Wait-die
    Wound-wait
Deadlocks

Even if nodes check WFG locally, **global** deadlocks are possible.

Local WFG:
No cycles

Local WFG:
No cycles
Deadlocks

Need to combine local WFG to discover global deadlocks

Local WFG: No cycles

E.g., at central detection node
Timestamp Ordering

Assign timestamp when transaction begins

If $ts(T_1) < ts(T_2) < \ldots < ts(T_n)$ then scheduler produces history equivalent to $T_1, T_2, \ldots, T_n$
Timestamp Ordering

Rule

If $p_i[x]$ and $q_j[x]$ are conflicting operations then

$p_i[x] <_S q_j[x]$ iff $ts(T_i) < ts(T_j)$
Timestamp Ordering

Example
Non-serializable schedule \( S \)

Node \( X \)

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

Node \( Y \)

\[
\begin{align*}
(T_2) & \quad d \leftarrow Y \\
(T_2) & \quad Y \leftarrow 2d \\
(T_1) & \quad b \leftarrow Y \\
(T_1) & \quad Y \leftarrow b+100
\end{align*}
\]
Timestamp Ordering

Example
Non-serializable schedule $S$

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>$(T_1) \quad b \leftarrow Y$</td>
</tr>
<tr>
<td>(T$_2$) $X \leftarrow 2c$</td>
<td>$(T_1) \quad Y \leftarrow b+100$</td>
</tr>
</tbody>
</table>
Timestamp Ordering

Example
Non-serializable schedule $S$

$t_s(T_1) < t_s(T_2)$

<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>

$ts(T_1) < ts(T_2)$

<table>
<thead>
<tr>
<th>abort $T_1$</th>
<th>abort $T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort $T_2$</td>
<td>abort $T_2$</td>
</tr>
</tbody>
</table>
Strict TO

Lock written items until it is certain that writing transaction has been successful

→ Avoid cascading aborts
Strict TO

Example
Non-serializable schedule \( S \)

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

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

\( (T_2) \) c ← X  delay

\( (T_1) \) b ← Y  reject!

abort \( T_1 \)
## Strict TO

### Example
Non-serializable schedule $S$

<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>
</tbody>
</table>

- $ts(T_1) < ts(T_2)$
- abort $T_1$
- reject $T_1$
- delay

$(T_2)$ $c \leftarrow X$

$(T_2)$ $X \leftarrow 2c$

---

$t(s(T_1)) < t(s(T_2))$
TO Scheduler

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>data item</td>
</tr>
<tr>
<td>mtsR[X]</td>
<td>maximum timestamp of a transaction that read X</td>
</tr>
<tr>
<td>mtsW[X]</td>
<td>maximum timestamp of a transaction that wrote X</td>
</tr>
<tr>
<td>nR[X]</td>
<td>number of transactions currently reading X (= 0, 1, 2, ...)</td>
</tr>
<tr>
<td>nW[X]</td>
<td>number of transactions currently writing X (= 0 or 1)</td>
</tr>
</tbody>
</table>
TO Scheduler

Part 1

\( r_i[X] \) arrives

if \( ts(T_i) < mtsw[X] \) then abort \( T_i \)
else
  if \( ts(T_i) > mtsr[X] \) then \( mtsr[X] \leftarrow ts(T_i) \)
  if queue is empty and \( nw[X] = 0 \) then
    \( nr[X] \leftarrow nr[X] + 1 \)
    initiate read \( X \)
  else add \( r_i[X] \) to queue
TO Scheduler

Part 2

\( w_i[X] \) arrives

if \( ts(T_i) < mtsw[X] \) or \( ts(T_i) < mtsr[X] \) then abort \( T_i \)
else

\[ mtsw[X] \leftarrow ts(T_i) \]

if queue is empty and \( nw[X] = 0 \) and \( nr[X] = 0 \) then

\[ nw[X] \leftarrow 1 \]

write \( X \) and wait for \( T_i \) to finish
else add \( w_i[X] \) to queue
TO Scheduler

Part 3

When some operation $o$ (read or write) on $X$ finishes

\[ \text{no}[X] \leftarrow \text{no}[X] - 1 \]
\[ \text{not} \_\text{done} \leftarrow \text{true} \]
\[ \text{while} \ \text{not} \_\text{done} \]
\[ o_j[X] \leftarrow \text{head of queue (with smallest timestamp)} \]
\[ \text{if } o = w \text{ and } \text{nr}[X] = 0 \text{ and } \text{nw}[X] = 0 \text{ then} \]
\[ \quad \text{remove } o_j[X] \text{ from queue} \]
\[ \ldots \]
TO Scheduler

Part 3

When some operation $o$ (read or write) on $X$ finishes

...  
  $nw[X] \leftarrow 1$
  write $X$ and wait for $T_j$ to finish

else if $o = r$ and $nw[X] = 0$ then
  remove $o_j[X]$ from queue
  $nr[X] \leftarrow nr[X] + 1$
  initiate read $X$

else not_done $\leftarrow$ true
TO Scheduler

For reads \( \text{nr}[X] \leftarrow \text{nr}[X] + 1; \text{ initiate read } X \)

For writes \( \text{nw}[X] \leftarrow 1; \text{ write } X \text{ and wait for } T_i \text{ to finish} \)

→ In part 3, the end of a write is only processed when all other writes for the transaction have completed
TO Scheduler

If a transaction is aborted, it must be retried with a new *larger* timestamp

\[
\text{mtsr}[X] = 10 \\
\text{mtsw}[X] = 9
\]

\[
\text{ts}(T') = 8 \\
\text{read } X
\]
TO Scheduler

If a transaction is aborted, it must be retried with a new *larger* timestamp

\[
\text{mtsr}[X] = 10 \\
\text{mtsw}[X] = 9
\]

\[
\text{ts}(T) = 8 \quad \text{starvation (keeps being aborted), should use } \text{ts}(T) = 11
\]

read X
TO Scheduler

Theorem

If $S$ is a schedule representing an execution by a TO scheduler then $S$ is serializable

Proof

Assume $T_i \rightarrow T_j$ in $P(S)$

$\Rightarrow \exists$ conflicting $p_i[X], q_j[X]$ in $S$: $p_i[X] <_S q_j[X]$

$\Rightarrow ts(T_i) < ts(T_j)$ by TO rule
TO Scheduler

Assume there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ in $P(S)$
\[\Rightarrow ts(T_1) < ts(T_2) < \ldots < ts(T_n) < ts(T_1), \text{which is a contradiction}\]

Hence, $P(S)$ is acyclic
\[\Rightarrow S \text{ is serializable}\]
Thomas Write Rule

$\text{mtsr}[X]$ \hspace{1cm} $\text{mtsw}[X]$

$\text{ts}(T_i)$

$T_i$ wants to write $X$
Thomas Write Rule

\( w_i[X] \) arrives

if \( ts(T_i) < mtsr[X] \) then abort \( T_i \)
else if \( ts(T_i) < mtsw[X] \) ignore write
else
    // as before

...
Thomas Write Rule

$T_i$ wants to write $X$

Why can't we let $T_i$ go ahead?
Thomas Write Rule

Why can’t we let $T_i$ go ahead?

mts$[X]$ is only the latest read—there could be others before
TO Optimizations

Update \texttt{mtsr} and \texttt{mtsw} when the action is executed, not when it is added to the queue.

\[ \texttt{mtsw}[X] = 9 \text{ or } 7? \]

<table>
<thead>
<tr>
<th>X</th>
<th>( w @ ts = 9 )</th>
<th>( w @ ts = 8 )</th>
<th>( w @ ts = 7 )</th>
</tr>
</thead>
</table>

active write
TO Optimizations

Use multiple versions of data

\[
\begin{align*}
X & \quad r_i[X] \quad ts(T_i) = 8 \\
\text{Value written @ ts = 9} & \\
\text{Value written @ ts = 7} & \\
\vdots & \\
\end{align*}
\]
## Timestamp Management

<table>
<thead>
<tr>
<th></th>
<th>data</th>
<th>mtsr</th>
<th>mtsw</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X_2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>X_n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tons of space and extra IO
Timestamp Management

Timestamp cache

<table>
<thead>
<tr>
<th>item</th>
<th>mtsr</th>
<th>mtsw</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>⋮</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If transaction reads/writes $X$, add/update entry for $X$ in cache

Periodically purge items $X$ with $\text{mtsr}[X] < \text{min}$, $\text{mtsw}[X] < \text{min}$

Track $\text{min}$ (e.g., choose $\text{min} \approx \text{current time} - d$)
Timestamp Management

Timestamp cache
Enforcing TO rule for $p_i[X]$:

if $X$ has entry in cache then
  use mtsr, mtsw values in cache
else
  assume mtsr[$X$] = mtsw[$X$] = min

Use hashing (on items) for cache
2PL ≠ TO

\[ T_1 \quad w_1[Y] \]
\[ T_2 \quad r_2[X] \quad r_2[Y] \quad w_2[Z] \quad ts(T_1) < ts(T_2) < ts(T_3) \]
\[ T_3 \quad w_3[X] \]

\[ S \quad r_2[X] \quad w_3[X] \quad w_1[Y] \quad r_2[Y] \quad w_2[Z] \]

S could be produced with TO but not with 2PL
2PL ≠ TO

Are all 2PL schedules TO?

any examples here?

previous example
Distributed TO Scheduler

node 1

scheduler 1

D_1

ts cache

D_2

ts cache

node 2

scheduler 2

...
Distributed TO Scheduler

Each scheduler is *independent*

Signal all schedulers involved at the end of the transaction
Pessimistic vs. Optimistic Control

Pessimistic

validate  read  (compute)  write

Optimistic

read  (compute)  validate  write
Pessimistic vs. Optimistic Control

Optimistic control enables more parallelism
Summary

2PL
Useful in a distributed system
Most popular
Deadlocks still possible

TO
Useful in a distributed system
Aborts more likely
No deadlocks