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

Properties
- Atomicity
- Consistency
- Isolation
- Durability

Concurrency control

Concurrent 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

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

Constraint: \(X = Y\)

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

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 \(\ll\) where:

1) \(S = \bigcup_i T_i\)

2) \(\ll \supset \bigcup_i \ll_{T_i}\)

3) For any two conflicting operations \(p, q \in S\), either \(p \ll q\) or \(q \ll p\) On same data, at least one is a write
Schedule

Example

\[ T_1 \rightarrow r_1[X] \rightarrow w_1[X] \]
\[ T_2 \rightarrow r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X] \]
\[ T_3 \rightarrow r_3[X] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z] \]
\[ S \rightarrow r_1[X] \rightarrow w_1[X] \rightarrow w_1[Y] \rightarrow w_1[Z] \]
\[ r_1[X] \rightarrow w_2[Y] \rightarrow w_1[X] \]

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

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

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

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>$(T_1)a \leftarrow X$</td>
<td>$(T_1)a \leftarrow X$</td>
</tr>
<tr>
<td>$2_{RX}(T_2)X \leftarrow a+100$</td>
<td>$5^{RY}(T_2)d \leftarrow Y$</td>
</tr>
<tr>
<td>$3^{RY}(T_2)c \leftarrow X$</td>
<td>$6_{RY}(T_2)Y \leftarrow 2d$</td>
</tr>
<tr>
<td>$4_{RX}(T_2)X \leftarrow 2c$</td>
<td>$7^{RY}(T_2)b \leftarrow Y$</td>
</tr>
<tr>
<td></td>
<td>$8_{RY}(T_1)Y \leftarrow b+100$</td>
</tr>
</tbody>
</table>

If $X = Y = 0$ initially, $X = 200$ and $Y = 100$ at the end ≠ serializable

---

**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)
  - Replicate P(X) at all processors
Deadlocks

Remember, 2PL may lead to deadlocks

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

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

\[ \begin{array}{c}
T_1 \\
T_2
\end{array} \]
\[ \begin{array}{c}
T_1 \\
T_2
\end{array} \]

Local WFG:
No cycles

Local WFG:
No cycles

Deadlocks

Need to combine local WFG to discover global deadlocks

\[ \begin{array}{c}
T_1 \\
T_2
\end{array} \]

Local WFG:
No cycles

\[ \begin{array}{c}
T_1 \\
T_2
\end{array} \]

Local WFG:
No cycles

E.g., at central detection node

Timestamp Ordering

Assign timestamp when transaction begins

\[ \text{If } ts(T_1) < ts(T_2) < \ldots < ts(T_n) \text{ then scheduler produces history equivalent to } T_1, T_2, \ldots, T_n \]
Timestamp Ordering

Rule
If \( p[x] \) and \( q[x] \) are conflicting operations then \( p[x] < q[x] \) iff \( ts(T_i) < ts(T_j) \)

Example
Non-serializable schedule \( S \)

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

- Node \( Y \):
  - \( (T_1) d \leftarrow Y \)
  - \( (T_2) Y \leftarrow 2d \)
  - \( (T_3) b \leftarrow Y \)
  - \( (T_4) Y \leftarrow b+100 \)

Relationship:
\( ts(T_1) < ts(T_2) \)
Strict TO

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

→ Avoid cascading aborts

Example
Non-serializable schedule $S$

$\text{ts}(T_1) < \text{ts}(T_2)$

<table>
<thead>
<tr>
<th>Node X</th>
<th>Node Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(T_1)$ a ← X</td>
<td>$(T_3)$ d ← Y</td>
</tr>
<tr>
<td>$(T_1)$ X ← a+100</td>
<td>$(T_2)$ Y ← 2d</td>
</tr>
</tbody>
</table>
| $(T_2)$ c ← X | $(T_2)$ b ← Y | abort $T_1$ reject $T_1$
| delay       | delay      |

Example
Non-serializable schedule $S$

$\text{ts}(T_1) < \text{ts}(T_2)$

<table>
<thead>
<tr>
<th>Node X</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(T_1)$ a ← X</td>
</tr>
<tr>
<td>$(T_1)$ X ← a+100</td>
</tr>
<tr>
<td>$(T_2)$ c ← X delay</td>
</tr>
<tr>
<td>abort $T_1$</td>
</tr>
<tr>
<td>$(T_3)$ X ← 2c</td>
</tr>
</tbody>
</table>

Strict TO

Example
TO Scheduler

$X$ data item

$\text{mtsR}[X]$ maximum timestamp of a transaction that read $X$

$\text{mtsW}[X]$ maximum timestamp of a transaction that wrote $X$

$nR[X]$ number of transactions currently reading $X$ ($= 0, 1, 2, ...$)

$nW[X]$ number of transactions currently writing $X$ ($= 0$ or $1$)
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

- \( no[X] \leftarrow no[X] - 1 \)
- \( not\_done \leftarrow true \)
- while not\_done
  - \( oj[X] \leftarrow \) head of queue (with smallest timestamp)
  - if \( o = w \) and \( nr[X] = 0 \) and \( nw[X] = 0 \) then
    - remove \( o[X] \) from queue
  - ...
TO Scheduler

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

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

\( \rightarrow \) 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

\( mtsr[X] = 10 \)
\( mtsw[X] = 9 \)
\( ts(T) = 8 \)
read \( X \)

TO Scheduler

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

\( mtsr[X] = 10 \)
\( mtsw[X] = 9 \)
\( ts(T) = 8 \)
read \( X \)

\( \rightarrow \) starvation (keeps being aborted), should use \( ts(T) = 11 \)

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 \( S; p_i[X] \leftarrow q_j[X] \)
\( \Rightarrow ts(T_i) < ts(T_j) \) 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

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

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

\[ T_i \text{ reads } X \]
\[ \text{ts}(T_i) \]
\[ \text{mts}(X) \]
\[ T_i \text{ wants to write } X \]

Why can’t we let \( T_i \) go ahead?
\( \text{mts}(X) \) is only the latest read—there could be others before

TO Optimizations

Update \( \text{mts} \) and \( \text{msw} \) when the action is executed, not when it is added to the queue
\[ \text{msw}(X) = 9 \text{ or } 7? \]

\[ \begin{array}{c}
X \\
\mid \mid \mid \\
\mid \mid \mid \\
\mid \mid \mid \\
\mid \mid \mid \\
\mid \mid \mid \\
\end{array} \]

\( \uparrow \) active write

TO Optimizations

Use multiple versions of data

\[ \begin{array}{c}
X \\
\mid \mid \mid \\
\mid \mid \mid \\
\mid \mid \mid \\
\mid \mid \mid \\
\mid \mid \mid \\
\end{array} \]

\[ r_i(X) \text{ ts}(T_i) = 8 \]

\[ \begin{array}{c}
\text{Value written } @ \text{ts} = 9 \\
\text{Value written } @ \text{ts} = 7 \\
\end{array} \]

Timestamp Management

\[ \begin{array}{c|c|c}
X_1 & \text{data} & \text{mts} \\
X_2 & & \\
& & \\
X_n & & \\
\end{array} \]

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>i</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 mtsr[X] < min, mtsw[X] < min

Track min (e.g., choose min ≈ current time – d)

---

**2PL ≠ TO**

\[
\begin{align*}
T_1 & \quad w[ Y ] \\
T_2 & \quad r_2[ X ] \mid r_2[ Y ] \mid w_2[ Z ] \\
T_3 & \quad w_3[ X ] \\
S & \quad r_3[ X ] \mid w_3[ X ] \mid w_3[ Y ] \mid r_3[ Y ] \mid w_3[ Z ]
\end{align*}
\]

S could be produced with TO but not with 2PL

---

**2PL ≠ TO**

Are all 2PL schedules TO?

Any examples here?

2PL schedules

TO schedules

previous example
Distributed TO Scheduler

Each scheduler is independent
Signal all schedulers involved at the end of the transaction

Pessimistic vs. Optimistic Control

Optimistic control enables more parallelism
<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PL</td>
</tr>
<tr>
<td>Useful in a distributed system</td>
</tr>
<tr>
<td>Most popular</td>
</tr>
<tr>
<td>Deadlocks still possible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useful in a distributed system</td>
</tr>
<tr>
<td>Aborts more likely</td>
</tr>
<tr>
<td>No deadlocks</td>
</tr>
</tbody>
</table>