Transaction Management

- Last time: Logging and Crash Recovery
- Today: Concurrency Control
- Requirements for transactions T to maintain DB consistency (ACID properties):
  - Atomicity: users perceive T as atomic, i.e., all or none of the effects are carried out (system)
  - Consistency: each T run by itself preserves consistency (user)
  - Isolation: "local transaction semantics", i.e., user can understand T as if executed in isolation, although DBMS executes T's concurrently (system)
  - Durability: after successful termination of T, effects are persistent even after system crash (as covered by the failure model) (system)

Concurrence Control

- Multiple transactions run in parallel
- if they run in isolation they are correct
- but what if they run concurrently?
- when does the schedule of execution of some transactions produce a consistent DB?

Example: T1, T2 with constraint A=B

<table>
<thead>
<tr>
<th>Serial Schedule S1</th>
<th>Serial Schedule S2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read(A)</code></td>
<td><code>read(A)</code></td>
</tr>
<tr>
<td><code>A=10</code></td>
<td><code>A=10</code></td>
</tr>
<tr>
<td><code>B=2</code></td>
<td><code>B=2</code></td>
</tr>
<tr>
<td><code>write(B)</code></td>
<td><code>write(B)</code></td>
</tr>
<tr>
<td><code>*25</code></td>
<td><code>*25</code></td>
</tr>
</tbody>
</table>

Both serial schedules ok (by assumption) and leave DB in a consistent state.

Serializable and Non-Serializable Schedules

<table>
<thead>
<tr>
<th>Bad Schedule</th>
<th>Good Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read(A)</code></td>
<td><code>read(A)</code></td>
</tr>
<tr>
<td><code>A=10</code></td>
<td><code>A=10</code></td>
</tr>
<tr>
<td><code>B=2</code></td>
<td><code>B=2</code></td>
</tr>
<tr>
<td><code>write(B)</code></td>
<td><code>write(B)</code></td>
</tr>
<tr>
<td><code>*25</code></td>
<td><code>*25</code></td>
</tr>
</tbody>
</table>

A simple test of correctness: Can we swap the statements of the schedule so that we produce an equivalent serial schedule? If yes, the schedule is a serializable one, i.e., a “good” one.
Conflicting Actions

- **Transaction** = sequence of actions like Read and Write: ri(X), wi(X) (transaction i reads/writes X)
- Let X≠Y. Swap OK:
  - ri(X); rj(Y) (even if X≠Y)
  - ri(X); wi(Y) and wi(X); rj(Y)
  - wi(X); wi(Y)
- Swap not OK (=conflicting actions):
  - two actions of the same transaction
  - ri(X); wi(X) and wi(X); rj(X)
- Def: S1, S2 are conflict equivalent if S1 can be transformed into S2 using swaps of non-conflicting actions
- Def: S is conflict serializable if it is conflict equivalent to a serial schedule

Good Schedule
Read(A); A := A + 100
Write(A)
Read(B); B := B + 100
Write(B)
Read(A); A := 2 * A
Write(A)
Read(B); B := 2 * B
Write(B)

Bad Schedule
Read(A); A := A + 100
Write(A)
Read(A); A := 2 * A
Write(A)
Read(B); B := B + 100
Write(B)
Read(B); B := 2 * B
Write(B)

How to detect whether S is conflict serializable?

- **Precedence graph** P(S) of schedule S
- **Nodes(S)** = transactions
- **Edges(S)**: Ti → Tj ("Ti takes precedence over Tj") whenever
  - p(X), q(X) are actions in S (Read or Write)
  - p(X) precedes q(X) in S
  - at least one of p(X), q(X) is write

Good Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>A := A + 100</td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td>B := B + 100</td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td>A := 2 * A</td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td>B := 2 * B</td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>

Bad Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>A := A + 100</td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td>B := B + 100</td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td>B := 2 * B</td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>

Testing for Serializability

- P(S) := \{(T1 \rightarrow T2)\}
Testing for Serializability

Lemma: S1, S2 conflict equivalent => P(S1) = P(S2)
Proof: if not P(S1) = P(S2) then there is an edge Ti -> Tj in S1 but not in S2:
=> S1 = ... pi(A) ... qj(A) ... and S2 = ... qj(A) ... pi(A) ...
=> S1, S2 are not conflict equivalent!

• Exercise: show that "<=" does not hold

Theorem: P(S) acyclic <=> S conflict serializable
Proof: if S conflict serializable then ex. serializable, conflict equivalence S' => P(S')=P(S'); note: P(S') is acyclic.
Conversely: P(S) is acyclic => it can be topologically sorted => serial schedule S1

How to Enforce Serializable Schedules?

• (Non-)Option1: run system; record P(S); check afterwards if P(S) was acyclic...

• Option2: Scheduler prevents P(S)'s cycles from occurring

• Two new actions:
  - lock(exclusive): li(X)
  - unlock: ui(X)

• Enforcing serializable schedules
  1) Well-formed transactions:
     Ti: ... li(X) ... pi(X) ... ui(X) ... no unlocks ... no locks
  2) Legal schedules:
     S ... li(X) ... pi(X) ... ui(X) ... "no if X"^n
  3) Two phase (2PL) locking
     Ti: ... li(X) ... pi(X) ... no unlocks ... no locks

if a lock request violates
  protocol the requesting
  transaction goes in wait state

Good Schedule
T1
Read(A);
A:=A+100
Write(A)
Read(A)
A:=2^A
Write(A)
Read(B)
B:=B+100
Write(B)
Read(B)
B:=2*B
Write(B)

Bad Schedule
T1
Read(A);
A:=A+100
Write(A)
Read(A)
A:=2^A
Write(A)
Read(B)
B:=B+100
Write(B)
Read(B)
B:=2*B
Write(B)
Exercise

• What schedules are legal?
• What transactions are well-formed?
  – S1 = l1(A) r1(B) w1(A) l1(B) u1(A) u1(B)
    r1(B) w1(B) u1(B) l1(B) u1(B)
  – S2 = l1(A) r1(A) w1(B) u1(A) u1(B)
    l1(B) r1(B) w1(B) u1(B)
  – S3 = l1(A) r1(A) u1(A) l1(B) w1(B) u1(B)
    l1(B) w1(B) r1(B) u1(B)

Two well-formed transactions on a non 2PL scheduler

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A)</td>
<td>Read(A); A := A + 100</td>
</tr>
<tr>
<td>Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>l1(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

Two non-serializable 2PL versions of the transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A)</td>
<td>Read(A); A := A + 100</td>
</tr>
<tr>
<td>Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>l1(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

Two serializable 2PL versions of the transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A)</td>
<td>Read(A); A := A + 100</td>
</tr>
<tr>
<td>Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>l1(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

Two serializable 2PL versions of the transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A)</td>
<td>Read(A); A := A + 100</td>
</tr>
<tr>
<td>Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>l1(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

Two non-serializable 2PL versions of the transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A)</td>
<td>Read(A); A := A + 100</td>
</tr>
<tr>
<td>Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>l1(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

Two serializable 2PL versions of the transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A)</td>
<td>Read(A); A := A + 100</td>
</tr>
<tr>
<td>Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>l1(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

Two non-serializable 2PL versions of the transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A)</td>
<td>Read(A); A := A + 100</td>
</tr>
<tr>
<td>Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>l1(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>
Theorem: S is 2PL (rules 1,2,3, s.156) => S conflict serializable
First: "shrink" transactions T to a point in time Sh(T), i.e., when they do their first unlock.
Lemma: if Ti \rightarrow Tj in P(S) then Sh(Ti) \leq Sh(Tj).
Proof if Ti \rightarrow Tj then
S = \ldots p(A) \ldots q(A) \ldots \ p.q conflicting actions
By rules 1,2:
S = \ldots p(A) \ldots u(A) \ldots l(A) \ldots q(A) \ldots
By rule 3: Sh(Ti) \leq Sh(Tj)

Lemma: if Ti \rightarrow Tj in P(S) then Sh(Ti) < Sh(Tj).
Proof if Ti \rightarrow Tj then
S = \ldots p(A) \ldots q(A) \ldots \ p.q conflicting actions
By rules 1,2:
S = \ldots p(A) \ldots u(A) \ldots l(A) \ldots q(A) \ldots
By rule 3: Sh(Ti) < Sh(Tj)

Theorem: S is 2PL (rules 1,2,3, s.156) => S conflict serializable
Proof Assume S is 2PL but not conflict serializable
=> P(S) is cyclic: T1 \rightarrow T2 \rightarrow \ldots \ Tn \rightarrow T1
=> (Lemma): Sh(T1) < Sh(T2) < \ldots < Sh(T1).
Contradiction!
=> P(S) is acyclic => S conflict serializable.

Fact: There are serializable schedules that cannot be produced by a 2PL scheduler:
**2PL Can Cause Deadlocks**

- Correctness: a 2PL transaction can be thought of as executing "instantaneously" at the time of the first unlock.
- But: 2 PL + sequence of lock requests => risk of deadlocks!
- Solutions:
  - detection: after timeout roll back (abort/restart) transaction
  - prevention: impose an order for locking
  - waits-for graph: detect or prevent cycles (and thus deadlocks)

```
T1                        T2
l1(A); Read(A);        l1(B); Read(B);  
A:=A+100              R:=2*B
Write(A);             Write(B);
l2 (B); Read(B)        l1(B); T1 WAIT
B:=2*B                l1(A); T1 WAIT
Write(B);
```

**Waits-For Graph**

Waits-for graph \( W(S) \) of schedule \( S \):
- **Nodes(\( S \))** = transactions holding or waiting for a lock
- **Edges(\( S \))**: \( T \rightarrow U \) "\( T \) waits for \( U \)" (to unlock...)
  - \( T \) waits for a lock \( l(A) \) that \( U \) holds, and
  - \( T \) cannot get \( l(A) \) unless \( U \) unlocks \( A \)

\( S \) has a deadlock \( \iff \) \( W(S) \) is cyclic:
- transactions in a cycle are deadlocked;
- if \( W(S) \) is acyclic, some transactions have no incoming edges
  - these can complete
  - further transactions have no in-edges, etc.

**Deadlock Prevention with Waits-For Graph**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(A); r(A)</td>
<td>l(C); r(C)</td>
<td>l(B); r(B)</td>
<td>l(D); r(D)</td>
<td></td>
</tr>
<tr>
<td>l(A); WAIT</td>
<td>l(C); WAIT</td>
<td>l(A); WAIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l(B); WAIT =&gt; DEADLOCK =&gt; don’t allow WAIT =&gt; cycle is prevented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DEADLOCK
Deadlock Prevention by Ordering Elements

- There is an order on the database elements: $A_1 < A_2 < \ldots < A_n$ (e.g., physical address of disk blocks)
- Transactions can request locks only in that order
  $\Rightarrow$ Deadlocks cannot occur:
  Assume $W(S)$ is cyclic: $T_1 \leftarrow T_2 \leftarrow T_3 \leftarrow \ldots \leftarrow T_n \leftarrow T_1$
  $\Rightarrow T_2$ waits for some $A_1$ held by $T_1$. $T_3$ waits for some $A_2$ held by $T_2$ etc.
  $\Rightarrow A_2 < A_1$ (since $T_2$ holds $l(A_2)$ but waits for $l(A_1)$)
  $\Rightarrow$ (similarly) $A_3 < A_2 < A_1 < A_3$ contradiction!
  $\Rightarrow W(S)$ is acyclic $\Rightarrow$ no deadlock

Deadlock Prevention by Timestamps

- Timestamp a transaction when it waits for a lock (keep this deadlock timestamp even for rollback)
- WAIT-DIE (wait for younger)
  - T older: $T \rightarrow U$: let the older $T$ wait
  - U older: $U \leftarrow T$: kill (rollback) the younger, waiting $T$
- WOUND-WAIT: (wait for older)
  - T older: $T \rightarrow U$: T "wounds" U; rollback younger U
  - U older: $U \leftarrow T$: let younger T wait
- These policies also prevent starvation: older transactions can kill younger ones, but the killed ones keep their timestamp on restart ($\Rightarrow$ will become oldest eventually)

Shared Locks

- So far: exclusive locks only, i.e., T locks before reading since T may write
  $\Rightarrow$ more restrictive than necessary since two reads are ok:
  ... $l_t(A) r_t(A) u_t(A) \ldots$ $l_t(A) r_t(A) u_t(A)$ (T1 or T2 has to wait)
  $\Rightarrow$ use shared locks for concurrent reading on the same object:
  ... $l_s(A) r_s(A) l_t(A) r_t(A) \ldots$ $u_s(A) u_t(A)$ (T1, T2 need not wait)
- Locking operations
  - lock shared/exclusive $l_s(A) / l_x(A)$
  - unlock shared/exclusive $u_s(A) / u_x(A)$
  - unlock (whatever): $u_t(A)$
Shared Locks

• (1) Well-formed transactions
  – read ok for shared locks: $T_i = \ldots$ $r_i(A) \ldots u_i(A) \ldots$
  – write/read ok for exclusive locks: $T_i = \ldots$ $x_i(A) \ldots w_i(A) \ldots$

• What about transactions that read and write the same object $A$?
  – option 1: request exclusive lock $lx(A)$ in advance
  – option 2: upgrade (need to read, but don’t know about write):
    $T_i = \ldots$ $r_i(A) \ldots u_i(A) \ldots$ $lx_i(A)$ $\ldots w_i(A)$ $wx_i(A)$

  this is like:
  • (i) a 2nd (exclusive) lock on $A$,
  • (ii) or unlock shared; lock exclusive

Shared Locks (cont’d)

• (1) Well-formed transactions
• (2) Legal schedules
  – $S = \ldots$ $ls(A) \ldots u(A) \ldots$
  – no $ls(A)$
  – $S = \ldots$ $lx(A) \ldots u(A) \ldots$ $no$ $lx(A)ls(A)$

• (3) 2PL as before but for upgrades:
  – upgrades are allowed in the growing phase (in both “views” (i), (ii))
  – common deadlock situation:
    $T_1 = ls(A) \ldots$ $lx(A)$
    $T_2 = ls(A) \ldots$ $lx(A)$
    $S = \ldots$ $lx(A)$ $\ldots$ $u(A)$ $\ldots$
  – solution: update locks $ul(A)$
    – can upgrade from $lu(A)$ to $lx(A)$ only, but not from $ls(A)$ to $lx(A)$:
      $T_1 = lu(A) \ldots$ $lx(A)$
      $T_2 = lu(A)$ (WAIT) $\ldots$ $lx(A)$

Increment Locks

• Atomic increment action
  $INC(A,k) := \{ \text{Read}(A); A := A + k; \text{Write}(A) \}$

• $INC(A,k)$, $INC(A,y)$ do not conflict

  $A = 7$
  $A = 5 \quad A = 15$
  $A = 17$

  • Consider schedules with increment lock $li(A)$
    – increment lock allows just that, for no read or write:
      e.g., the following can execute without delay:
      $T_1 = ls_1(A); r_1(A); li_1(B); inc_1(B,k); u_1(A); u_1(B)$
      $T_2 = ls_2(A); r_2(A); li_2(B); inc_2(B,k); u_2(A); u_2(B)$

  $A = 7$
  $A = 5 \quad A = 15$
• Conflict serializability can be too restrictive. E.g., schedule $S_1 = T_1 = r_1(A); w_1(A)$
  $T_2 = w_2(A)$;
  $T_3 = w_3(A)$
precedence graph $P(S_1) = T_1 \rightarrow T_2 \rightarrow T_3$
$\Rightarrow S_1$ is not conflict serializable!
• but compare $S_1$ with the following serial schedule $S_2$
  $S_1 = r_1(A); w_2(A); w_1(A); w_3(A)$ (not conflict serializable)
  $S_2 = r_1(A); w_1(A); w_2(A); w_3(A)$ ("view equivalent" serial schedule)
• crux: the writes $w_1(A); w_2(A)$ are irrelevant for the outcome, so $S_1$ and $S_2$ have the same result

---

**View Serializability**

- **Def.:** Schedules $S_1, S_2$ are **view equivalent** if
  (V1) if in $S_1$: $w_j(A) \Rightarrow r_i(A)$ "$r_i(A)$ reads value produced by $w_j(A)$"
  then in $S_2$: $w_j(A) \Rightarrow r_i(A)$
  (V2) if in $S_1$: $r_i(A)$ reads initial DB value,
  then in $S_2$: $r_i(A)$ also reads initial DB value
  (V3) if in $S_1$: $T_i$ does last write on $A$,
  then in $S_2$: $T_i$ also does last write on $A$
- **Def.:** Schedule $S_1$ is **view serializable** if it is view equivalent to
  some serial schedule
- **Example:**
  $S_1 = r_1(A); w_2(A); w_1(A); w_3(A)$ (not conflict serializable)
  $S_2 = r_1(A); w_1(A); w_2(A); w_3(A)$ ("view equivalent" serial schedule)

---

**View Serializability**

**Lemma:** $S$ conflict serializable implies $S$ view serializable

**Proof:** swapping non-conflicting actions
  - does not change what transactions read
  - does not change final DB state

- So: conflict serializable $\subseteq$ view serializable $\subseteq$ all schedules
  (note: these inclusions are strict)
- view serializable schedules that are not conflict serializable involve useless write, i.e., as situation like $S = w_2(A) \ldots w_3(A) \ldots \Rightarrow$no reads$\Rightarrow$
Testing for View Serializability

- (1) Add final transaction \( T_f \) that reads all DB (takes care of V2)
  
  \[ S = \ldots w1(A) \ldots w2(A) \ldots r_f(A) \]

- (2) Add initial transaction \( T_b \) that writes all DB (takes care of V3)
  
  \[ S = w_b(A) \ldots r_f(A) \ldots w2(A) \ldots \]

- (3) Create labeled precedence graph of \( S \):
  
  - (3a) If \( w_i(A) \Rightarrow r_j(A) \) in \( S \), add \( T_i \rightarrow T_j \)
  
  - (3b) For each \( w_i(A) \Rightarrow r_j(A) \) consider each \( w_k(A) \) \( k \neq b, i, j \)
    
    - If \( T_i = T_b \land T_j = T_f \) then insert \( T_k \rightarrow T_j \) (for some new \( p \))
    - If \( T_i = T_b \land T_j \neq T_f \) then insert \( T_j \rightarrow T_k \)
    - If \( T_i \neq T_b \land T_j = T_f \) then insert \( T_k \rightarrow T_i \)

- (4) Check if \( LP(S) \) is "acyclic" (if so, \( S \) is view serializable)
  
  - For each pair of "p" arcs (\( p \neq 0 \)), choose one

Testing for View Serializability: Example

Example: check if \( Q \) is V-S:

\[ Q = [ \begin{array}{c} \text{r}_1(A) \ w_2(A) \ w_1(A) \ w_3(A) \end{array} ] \]

\[ Q' = w_b(A) \Rightarrow r_1(A) \ w_2(A) \Rightarrow r_3(A) \ w_1(A) \Rightarrow r_f(A) \]

\[ w_2, w_3 \text{ after } r_1 \]

\[ w_2, w_1 \text{ before } w_3 \]

\[ LP(S) \text{ acyclic!} \]

\( S \text{ is V-S} \)

Testing for View Serializability: Another Example

\[ Z = w_b(A) \Rightarrow r_1(A) \ w_2(A) \Rightarrow r_1(A) \ w_1(A) \Rightarrow r_f(A) \]

\[ w_1 \text{ after } r_3 \]

or \( w_1 \text{ before } w_2 \)

\[ LP(Z) \text{ acyclic, so } Z \text{ is V-S} \]

(\( Z + S \) does indeed do same thing)