Concurrency Control

$T_1 \quad T_2 \quad \ldots \quad T_n$

DB
(consistency constraints)
Transactions

A transaction is an execution of a program having the following properties:

Atomicity
A transaction either happens or doesn’t. A transaction either completes and the results become visible or no results are visible.

Consistency
Transactions preserve correctness of database.

Isolation
Each transaction is unaware of other transactions executing concurrently

Durability
The results of a completed transaction are permanently installed within D.B.

ACID properties
A transaction has exactly one of two possible outcomes:

a) **Commit**
   Program execution completes and the results become permanent in database.

b) **Abort**
   Program execution was not successful. “Results” are not installed into database.

- Transactions may abort due to hardware failure, system error, bad input data, or as a means to ensure consistency.

---

**Example:**

T1: \( \text{Read}(A) \quad \text{Write}(A) \quad \text{Read}(B) \quad \text{Write}(B) \)

\( A \leftarrow A+100 \quad A \leftarrow A \times 2 \quad B \leftarrow B+100 \quad B \leftarrow B \times 2 \)

Constraint: \( A = B \)
Main idea of concurrency control

1. An execution without any interleaving is OK
   serial: $T_1 ; T_2$ or $T_2 ; T_1$

2. If an execution has the same effect as a serial execution then it is also acceptable serializable

Main goal of concurrency control:

guarantee serializability

---

Serial Schedule A (“good” by definition)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
Serial Schedule B (equally “good”)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Read(A); A ← A+100</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Read(A); A ← A+100
Write(A);
Read(B); B ← B+100;
Write(B);

Interleaved Schedule C (good because it is equivalent to A)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>250</td>
</tr>
</tbody>
</table>

A and C are equivalent because if they start
from same initial values they end up with same results
**Interleaved Schedule D (bad!)**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A×2;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

**Schedule E (good by “accident”)**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2’</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A×1;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td>125</td>
</tr>
</tbody>
</table>

The accident being the particular semantics

Same as Schedule D but with new T2’
• Want schedules that are “good”, i.e., equivalent to serial regardless of
  – initial state and
  – transaction semantics
• Only look at order of read and writes
What we say to a database

“delete all movies not
directed by Berto”

read, read, write, read, read…. 

What the database hears

Example of a read/write schedule:
SC=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

Definition

S₁, S₂ are conflict equivalent schedules
if S₁ can be transformed into S₂ by a
series of swaps of adjacent non-
conflicting actions.

Non-conflicting actions:
• actions on different data
• read/read on the same data
Definition

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

Example:

$SC = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

$SC' = r_1(A)w_1(A) r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

$T_1$  $T_2$
However, for SD:
$SD = r_1(A)w_1(A)r_2(A)w_2(A)\ r_2(B)w_2(B)r_1(B)w_1(B)$

- as a matter of fact, $T_2$ must precede $T_1$ in any equivalent schedule, i.e., $T_2 \rightarrow T_1$
- And vice versa

$\Rightarrow$ SD cannot be rearranged into a serial schedule
$\Rightarrow$ SD is not “equivalent” to any serial schedule
$\Rightarrow$ SD is “bad”
Returning to SC

\[
\text{SC}= r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)
\]

\[
T_1 \rightarrow T_2 \\
T_1 \rightarrow T_2
\]

no cycles \(\Rightarrow\) SC is “equivalent” to a serial schedule
(in this case \(T_1,T_2\))

---

**Precedence graph** \(P(S)\) (\(S\) is schedule)

Nodes: transactions in \(S\)

Arcs: \(T_i \rightarrow T_j\) whenever

- \(p_i(A), q_j(A)\) are actions in \(S\)
- \(p_i(A) <_S q_j(A)\)
- at least one of \(p_i, q_j\) is a write

\(<_S\) : order of appearance in schedule
Exercise:

- What is $P(S)$ for $S = w_3(A) \ w_2(C) \ r_1(A) \ w_1(B) \ r_1(C) \ w_2(A) \ r_4(A) \ w_4(D)$

- Is $S$ serializable?

Lemma

$S_1$, $S_2$ conflict equivalent $\Rightarrow$ $P(S_1) = P(S_2)$

Proof:
Assume $P(S_1) \neq P(S_2)$

$\Rightarrow \exists T_i, T_j: T_i \rightarrow T_j$ in $S_1$ and not in $S_2$

$\Rightarrow S_1 = \ldots p_i(A) \ldots q_j(A) \ldots$

$S_2 = \ldots q_j(A) \ldots p_i(A) \ldots$

$\Rightarrow S_1, S_2$ not conflict equivalent
Lemma

$S_1, S_2$ conflict equivalent $\Rightarrow P(S_1) = P(S_2)$

Is the converse true?

A: yes  B: no

Note: $P(S_1) = P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Counter example:

$S_1 = \text{w}_1(A) \text{ r}_2(A) \text{ w}_2(B) \text{ r}_1(B)$

$S_2 = \text{r}_2(A) \text{ w}_1(A) \text{ r}_1(B) \text{ w}_2(B)$
Theorem

$P(S_1)$ acyclic $\iff S_1$ conflict serializable

$(\iff)$ Assume $S_1$ is conflict serializable

$\Rightarrow \exists S_s: S_s, S_1$ conflict equivalent

$\Rightarrow P(S_s) = P(S_1)$

$\Rightarrow P(S_1)$ acyclic since $P(S_s)$ is acyclic

$(\Rightarrow)$ Assume $P(S_1)$ is acyclic

Transform $S_1$ as follows:

(1) Take $T_1$ to be transaction with no incoming arcs

(2) Move all $T_1$ actions to the front

$S_1 = \ldots q_j(A) \ldots p_1(A) \ldots$

(3) we now have $S_1 = < T_1 \text{ actions }> <\ldots \text{ rest } \ldots>$

(4) repeat above steps to serialize rest!
How to enforce serializable schedules?

*Option 1:* run system, recording P(S); check for P(S) cycles and declare if execution was good; or abort transactions as soon as they generate a cycle.

*Option 2:* prevent P(S) cycles from occurring.

---

![Diagram showing transactions T1 through Tn interacting with Scheduler and DB.](image-url)
A locking protocol

Two new actions:
- lock (exclusive): \( l_i(A) \)
- unlock: \( u_i(A) \)

Rule #1: Well-formed transactions

\( Ti: \ ... l_i(A) \ ... p_i(A) \ ... u_i(A) \ ... \)
Rule #2  Legal scheduler

\[ S = \ldots \text{i}_i(A) \ldots \cdots \text{u}_i(A) \ldots \cdots \]
\[ \text{no } \text{l}_j(A) \]

Exercise:

- What schedules are correct?
  S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)
    r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)
  S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)
    l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)
  S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)
    l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)
Exercise:

- What schedules are correct?
  
  $S_1 = l_1(A)l_1(B)r_1(A)w_1(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

  $S_2 = l_1(A)r_1(A)w_1(B)u_1(A)l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

  $S_3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

C: S3 correct

Just having locks is not enough!

Example: this allows Schedule F (bad)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1(A); Read(A)$</td>
<td>$l_2(A); Read(A)$</td>
</tr>
<tr>
<td>$A \leftarrow A + 100; Write(A); u_1(A)$</td>
<td>$A \leftarrow A x 2; Write(A); u_2(A)$</td>
</tr>
<tr>
<td>$l_1(B); Read(B)$</td>
<td>$l_2(B); Read(B)$</td>
</tr>
<tr>
<td>$B \leftarrow B + 100; Write(B); u_1(B)$</td>
<td>$B \leftarrow B x 2; Write(B); u_2(B)$</td>
</tr>
</tbody>
</table>
Two phase locking (2PL)

\[ T_i = \ldots. l_i(A) \ldots . u_i(A) \ldots . \]

![Graph showing time, growing phase, shrinking phase, and number of locks held by Ti.](image)

37

38
2PL prevents Schedule F:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1(A); \text{Read}(A)$</td>
<td>$l_2(A); \text{Read}(A)$</td>
</tr>
<tr>
<td>$A \leftarrow A + 100; \text{Write}(A)$</td>
<td>$A \leftarrow Ax2; \text{Write}(A)$</td>
</tr>
<tr>
<td>$l_1(B); \text{u}_1(A)$</td>
<td>$\text{delayed}$</td>
</tr>
<tr>
<td>$\text{Read}(B); B \leftarrow B + 100$</td>
<td>$\text{Write}(B); \text{u}_1(B)$</td>
</tr>
</tbody>
</table>
2PL prevents Schedule F:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A); Read(A)</td>
<td>l₂(A); Read(A)</td>
</tr>
<tr>
<td>A ← A+100; Write(A)</td>
<td>A ← Ax2; Write(A)</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td>delayed</td>
</tr>
<tr>
<td>Read(B); B ← B+100</td>
<td></td>
</tr>
<tr>
<td>Write(B); u₁(B)</td>
<td></td>
</tr>
</tbody>
</table>

2PL may however result in deadlock:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A); Read(A)</td>
<td>l₂(B); Read(B)</td>
</tr>
<tr>
<td>A ← A+100; Write(A)</td>
<td>B ← Bx2; Write(B)</td>
</tr>
<tr>
<td>delayed</td>
<td>delayed</td>
</tr>
</tbody>
</table>

Usual fix: deadlocked transactions are aborted and rolled back
Next step:

Show that 2 Phase Locking $\implies$ conflict-serializable schedules

Theorem 2PL $\implies$ conflict serializable schedule

To help in proof:

Definition $\text{Shrink}(T_i) = \text{SH}(T_i) =$ first unlock action of $T_i$
Lemma
Ti → Tj in S ⇒ SH(Ti) <_S_ SH(Tj)

Proof of lemma:
Ti → Tj means that
S = ... pi(A) ... qj(A) ...; p,q conflict
By rules 1,2:
S = ... pi(A) ... ui(A) ... li(A) ... qj(A) ...
By 2PL: SH(Ti) SH(Tj)
So, SH(Ti) <_S_ SH(Tj)

Theorem 2PL ⇒ conflict serializable schedule

Proof:
(1) Assume P(S) has cycle
   T_1 → T_2 → ... T_n → T_1
(2) By lemma: SH(T_1) < SH(T_2) < ... < SH(T_1)
(3) Impossible, so P(S) acyclic
(4) ⇒ S is conflict serializable
How does locking work in practice?

• Every system is different

But here is one (simplified) way …

Sample Locking System:

(1) Don’t trust transactions to request/release locks
(2) Hold all locks until transaction commits

![Graph showing the number of locks over time]
What are the objects we lock?

- Relation A
- Relation B
- Tuple A
- Tuple B
- Tuple C
- Disk block A
- Disk block B
- DB
- DB
- DB
• Locking works in any case, but should we choose small or large objects?

• If we lock large objects (e.g., Relations)
  - Need few locks
  - Low concurrency

• If we lock small objects (e.g., tuples, fields)
  - Need more locks
  - More concurrency

• Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency....
  - Shared locks
  - Multiple granularity
  - Inserts, deletes
  - Other types of C.C. mechanisms
  - Weaker guarantees (e.g. in the cloud)
Shared locks

So far:
\[ S = \ldots l_1(A) \ r_1(A) \ u_1(A) \ \ldots \ l_2(A) \ r_2(A) \ u_2(A) \ \ldots \]

\[ \text{Do not conflict} \]

Instead:
\[ S=\ldots \ l{s_1}(A) \ r_1(A) \ l{s_2}(A) \ r_2(A) \ \ldots \ u{s_1}(A) \ u{s_2}(A) \]

Lock actions
\[ l-\text{t}_i(A): \text{lock} \ A \ \text{in} \ t \ \text{mode} \ (t \ \text{is} \ S \ \text{or} \ X) \]
\[ u-\text{t}_i(A): \text{unlock} \ t \ \text{mode} \ (t \ \text{is} \ S \ \text{or} \ X) \]

Shorthand:
\[ u_i(A): \text{unlock whatever modes} \]
\[ Ti \ \text{has locked} \ A \]
Rule #1  Well formed transactions

$T_i = \ldots l-S_1(A) \ldots r_1(A) \ldots u_1(A) \ldots$

$T_i = \ldots l-X_1(A) \ldots w_1(A) \ldots u_1(A) \ldots$

• What about transactions that read and write same object?

Option 1: Request exclusive lock

$T_i = \ldots l-X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots$
• What about transactions that read and write same object?

**Option 2: Upgrade**
(E.g., need to read, but don’t know if will write…)

\[ T_i = \ldots l-S_1(A) \ldots r_1(A) \ldots l-X_1(A) \ldots w_1(A) \ldots u(A) \ldots \]

Upgrade lock on A from shared to exclusive

---

**Rule #2  Legal scheduler**

\[ S = \ldots l-S_i(A) \ldots u_i(A) \ldots \]

\[ \text{no } l-X_j(A) \]

\[ S = \ldots l-X_i(A) \ldots u_i(A) \ldots \]

\[ \text{no } l-X_j(A) \]

\[ \text{no } l-S_j(A) \]
A way to summarize Rule #2

Compatibility matrix

<table>
<thead>
<tr>
<th>Comp</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Rule #3 2PL transactions
Same as before, with upgrades allowed in growing phase

Theorem  Rules 1,2,3 ⇒ Conf.serializable for S/X locks schedules

Proof: similar to X locks case
Multiple granularity:

Tree-based concurrency control

Example

• all objects accessed through root, following pointers

---

can we release A lock if we no longer need A??
Idea: traverse like “Monkey Bars”

Why does this work?

• Assume all Ti start at root; exclusive lock
• Ti \rightarrow Tj \Rightarrow Ti locks root before Tj

• Actually works if we don’t always start at root
Rules: tree protocol (exclusive locks)

(1) First lock by Ti may be on any item
(2) After that, item Q can be locked by Ti only if parent(Q) locked by Ti
(3) Items may be unlocked at any time
(4) After Ti unlocks Q, it cannot relock Q

• Tree-like protocols are used typically for B-tree concurrency control

E.g., during insert, do not release parent lock, until you are certain child does not have to split