Chapter 9  Concurrency Control

How to prevent harmful interference btw transactions?

=> scheduling techniques based on
- locks
- timestamps and validation

Correctness depends on scheduling of transactions

A schedule

- Chronological (possibly interleaving) order in which actions of transactions are executed
- A correct schedule is equivalent to executing transactions one-at-a-time in some order

Example:

Constraint: \( A = B \)

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

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

Schedule D

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

Schedule E

Same as Schedule D

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

Example: Sa (Schedule a) = r1(A)w1(A)r1(B)w1(B)r2(B)w2(B)
T1                                  T2

Example: (of swapping non-conflicting actions)
Sc=r1(A)w1(A)r2(A)w1(A)r1(B)w1(B)r2(B)w2(B)

Example: Sc'=r1(A)w1(A)r1(B)w1(B)r2(A)w2(A)r2(B)w2(B)
T1                                  T2

- A schedule is **serial**, if actions of transactions are not interleaved
  - e.g., (T1, T2) or (T2, T1)
  - A serial schedule obviously maintains consistency (assuming correctness of individual transactions)

- Could we reorder a schedule into an equivalent serial schedule?
  - Actions **conflict**, if swapping them may change the meaning of a schedule:
    - any two actions of a single transaction
    - two actions on a common DB element A, one of which is WRITE(A)

- Want schedules that are "good", regardless of
  - initial state ("good" in any DB state) and
  - transaction semantics

- Only look at order of **READs** and **WRITEs**
  - Note: transactions see values in buffers, not on disk => this time ignore INPUT/OUTPUTs
However, for Sd:

\[
Sd = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)
\]

- Sd cannot be rearranged into a serial schedule
- Sd is not "equivalent" to any serial schedule
- Sd is "bad"

Concepts

**Transaction**: sequence of \( r_i(x) \), \( w_i(x) \) actions

**Conflicting actions**: \( r_i(A) \) \( w_i(A) \) \( r_j(A) \) \( w_j(A) \)

If schedule S contains conflicting actions...
... \( p_1(A), \ldots, q_k(A), \ldots \) [i.e., one of \( p, q \) is \( w \)],
transaction \( T_h \) must precede \( T_k \) in a corresponding serial schedule. Denote this by \( T_h \rightarrow T_k \)

Returning to Sc

\[
Sc = r_1(A)w_1(A)r_1(A)w_1(B)r_2(B)w_2(B)
\]

\( T_1 \rightarrow T_2 \)

No cycles \( \Rightarrow \) Sc is "equivalent" to a serial schedule
(in this case \( T_1, T_2 \))

Definition

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

**NB**: Conflict serializability is a sufficient (but not a necessary) condition for serializability (equivalence to some serial schedule)

Easier to enforce than serializability, therefore generally assured by commercial systems

Definition

\( S_1, S_2 \) are **conflict equivalent** schedules if \( S_1 \) can be transformed into \( S_2 \) by a series of swaps on non-conflicting actions.

\( \Rightarrow \) effect of both \( S_1 \) and \( S_2 \) on the DB is the same

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

**Nodes**: transactions \( T_1, T_2, \ldots \) in S

**Arcs**: \( T_i \rightarrow T_j \) for \( i \neq j \) whenever
- \( pi(A), qj(A) \) are conflicting actions in S,
  (same element \( A \), at least one of actions is a write)
- action \( pi(A) \) precedes \( qj(A) \) in S
**Exercise:**

- What is $P(S)$ for $S = w_3(A) w_2(C) r_1(A) w_1(B) r_1(C) w_2(A) r_2(A) w_3(D)$?

- Is $S$ serializable?

**Lemma** Let $S_1$, $S_2$ be schedules for the same set of transactions.

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

**Proof:**

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

$\Rightarrow \exists T_i: T_i \rightarrow T_j$ in $P(S_1)$ and not in $P(S_2)$

$\Rightarrow S_1 = ... p(A)... q(A)... \quad \text{Conflict}$

$S_2 = ... q(A)... p(A)...$

$\Rightarrow S_1, S_2$ not conflict equivalent

Note: $P(S_1) = P(S_2) \iff S_1, S_2$ conflict equivalent

**Counter example:**

$S_1 = w_1(A) r_2(A) w_2(B) r_2(B)$

$S_2 = r_2(A) w_2(A) r_2(B) w_2(B)$

**Theorem**

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

$\iff$ Assume $S_1$ is conflict serializable

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

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

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

**Theorem (cont.)**

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

$(\Rightarrow)$ 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

$S_1 = ... q(X) ... p(A) ...$

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

4. repeat above steps to serialize rest!

**How to enforce serializable schedules?**

Option 1: (Optimistic strategy)

Run system, recording $P(S)$;

At end of day, check $P(S)$ for cycles, and declare if execution was good
How to enforce serializable schedules?

Option 2: (Pessimistic strategy)

Prevent occurrence of cycles in P(S)

\[
T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n
\]

Scheduler

DB

A locking protocol

Two new actions:

- lock (exclusive): \( l_i(A) \)
- unlock: \( u_i(A) \)

Rule #1: Well-formed transactions

\[ T_i: \ldots l_i(A) \ldots \pi_i(A) \ldots u_i(A) \ldots \]

- Lock elements \( A \) before accessing them
  
  \( \pi_i(A) \) is a read or a write
- Eventually, release the locks \( \pi_i(A) \)

Rule #2: Legal scheduler

\[ S = \ldots l_i(A) \ldots u_i(A) \ldots \]

- no \( l_i(A) \) for \( i \neq j \)

- At most one transaction \( T_i \) can hold a lock on any element \( A \)

Exercise:

- What schedules are legal?
  - What transactions are well-formed?

\[
S_1 = l_1(A)\, l_2(A)\, r_1(A)\, w_1(A)\, r_2(A)\, w_2(A)\, u_1(A)\, u_2(A)\]

\[
S_2 = l_1(A)\, r_1(A)\, w_1(A)\, r_2(A)\, w_2(A)\, u_1(A)\, u_2(A)\]

\[
S_3 = l_1(A)\, r_1(A)\, u_1(A)\, l_2(A)\, w_1(A)\, r_2(A)\, u_1(A)\]

\[
l_1(B)\, r_1(B)\, w_1(B)\, u_1(B)\]

\[
l_2(B)\, r_2(B)\, w_2(B)\, u_2(B)\]

Schedule F (with simple locking)

\[
T_1 \quad T_2
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

\[
l_1(A)\, ; \text{Read}(A)\]

\[
A := A + 100; \text{Write}(A)\, u_i(A)\]

\[
l_1(A)\, ; \text{Read}(A)\]

\[
A := 2x; \text{Write}(A)\, u_i(A)\]

\[
l_1(B)\, ; \text{Read}(B)\]

\[
B := B + 100; \text{Write}(B)\, u_i(B)\]

Constraint violation! 150 150
Simple-minded locking not sufficient to ensure serializability (i.e., correctness)!

→ More advanced protocol known as "two phase locking"

Rule #3 Two phase locking (2PL)

for transactions

\[ T_i = \ldots \ l(A) \ \ldots \ u(A) \ \ldots \]

no unlocks no locks

• All lock requests of a transaction have to precede its unlock requests

Schedule G (with 2PL)

\[
\begin{array}{ll}
T_1 & T_2 \\
\text{l(A); Read(A)} & \text{l(A); Read(A)} \\
A := A + 100; Write(A) & \text{delayed A := A + 100; Write(A); u(A);} \\
\text{Read(B); B := B + 100} & \text{Read(B); B := B + 100} \\
\text{Write(B); u(B);} & \text{Write(B); u(B);}
\end{array}
\]

Schedule H (T2 reversed)

\[
\begin{array}{ll}
T_1 & T_2 \\
\text{l(A); Read(A)} & \text{l(B); Read(B)} \\
A := A + 100; Write(A) & \text{delayed A := A + 100; Write(A); u(A);} \\
B := B \times 2; Write(B) & \text{delayed B := B \times 2; Write(B); u(B);}
\end{array}
\]

• Neither proceeds: a deadlock
  - System must rollback (= abort & restart) at least one of T1, T2

Next step:

Show that Rules #1,2,3 ⇒ conflict (2PL) serializable schedule
To help in proof:

**Definition**  \[ \text{Shrink}(T_i) = SH(T_i) = \text{first unlock action of } T_i \]

Lemma: Let S be a 2PL schedule.
\[ T_i \to T_j \text{ in } P(S) \Rightarrow SH(T_i) <_S SH(T_j) \]

**Proof of lemma:**

\[ T_i \to T_j \text{ means that} \]
\[ S = \ldots p(A) \ldots q_i(A) \ldots; \text{ p,q conflict} \]
By rules 1,2:
\[ S = \ldots p(A) \ldots u_i(A) \ldots l_i(A) \ldots q(A) \ldots \]
By rule 3:
\[ \text{SH}(T_i) \to \text{SH}(T_j) \]
So,  \[ SH(T_i) <_S SH(T_j) \]

**Theorem II**

Rules #1,2,3 (that is, 2PL) \[ \Rightarrow \] conflict serializable schedule

**Proof:** Let S be a 2PL schedule.
Assume P(S) has cycle
\[ T_1 \to T_2 \to \ldots T_n \to T_1 \]
By Lemma: \[ SH(T_i) < SH(T_2) < \ldots < SH(T_1) \]
Impossible, so P(S) acyclic
\[ \Rightarrow S \text{ is conflict serializable (by Th. I)} \]

**Shared lock**

So far only exclusive locks:
\[ S = \ldots l_1(A) \ u_1(A) \ u_2(A) \ u_3(A) \ldots \]

Do not conflict \[ \Rightarrow \] locking unnecessary

**Instead** use shared locks (S) for reading:
\[ S = \ldots l_1(A) \ r_1(A) \ l_1(A) \ r_2(A) \ u_1(A) \ u_2(A) \]

• Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency....
   - Shared locks
   - Multiple granularity
   - Inserts, deletes and phantoms
   - Other types of C.C. mechanisms
     • Timestamping
     • Validation

Write actions conflict \[ \Rightarrow \] use exclusive (X) locks for writing

**Lock actions:**
\[ l-m(A): \text{lock A in mode } m \text{ (S or X) for } T_k \]
\[ u_k(A): \text{release (whatever) lock(s) held by transaction } T_k \text{ on element } A \]
**Rule #1**  Well-formed transactions

\[ T_i = \ldots \text{S lock for reading} \ldots \text{X lock for writing} \ldots \]

- Request
  - an S-lock for reading
  - an X-lock for writing
- Release the locks eventually

**Rule #2**  Legal scheduler

\[ S = \ldots \text{S lock for reading} \ldots \text{X lock for writing} \ldots \]

- no X for \( j \neq i \)

\[ S = \ldots \text{X lock for reading} \ldots \text{S lock for writing} \ldots \]

- no X for \( j \neq i \)
- no S for \( j \neq i \)

**Rule #3**  (2PL)

Only change to previous:
Lock upgrades
\[ S(A) \rightarrow \{S(A), X(A)\} \] or \[ S(A) \rightarrow X(A) \]
are allowed only in the growing phase

**Theorem**  Rules 1,2,3 \( \Rightarrow \) Conf.serializable

for S/X locks schedules

**Proof:**  Similar to the X locks case

**Lock types beyond S/X**
Examples:
1. update lock
2. increment lock  (see the textbook)
**Update locks**

A common deadlock problem with upgrades:

\[
\begin{align*}
T_1 & \quad T_2 \\
\text{I-S}(A) & \quad \text{I-S}(A) \\
\text{I-X}(A) & \quad \text{I-X}(A) \\
\end{align*}
\]

--- Deadlock ---

**Solution**

If \(T_i\) wants to read \(A\) and knows it may later want to write \(A\), it requests an update lock (not shared).

**Note:** object \(A\) may be locked in different modes at the same time...

\[
S_1=\ldots\text{I-S}(A)\ldots\text{I-S}(A)\ldots\text{I-U}(A)\ldots
\]

To grant a lock in mode \(m\), mode \(m\) must be compatible with all currently held locks on object.

**How does locking work in practice?**

- Every system is different
- Here is one (simplified) way ...

**Sample Locking System:**

1. Don't trust transactions to request/release locks
2. Do not release locks until transaction commits/aborts:
Read(A), Write(B)
(l(A), Read(A), l(B), Write(B)…
Read(A), Write(B)

Scheduler, part I
Insert appropriate lock requests
Scheduler, part II
Execute or delay, based on existing locks
Read(A), Write(B)

Lock table
Every possible object

Conceptually
If null, object is unlocked

Lock info for A - example

Element: A
Group mode: U
Waiting: yes

T1: R, ho
T2: U, ho
T3: X, yes

To other lock table entries of transaction T3

What are the objects we lock?

Relation A
Relation B
Disk block
Disk block

DB
DB
DB

But use hash table:

If object not found in hash table, it is unlocked

Lock info for A - example

- Locking works in any case, but should we choose small or large objects?
- If we lock large objects (e.g., Relations)
  - Need few locks
  - Get low concurrency
- If we lock small objects (e.g., tuples, fields)
  - Need more locks (=> overhead higher)
  - Get more concurrency
We can have it both ways!!

Ask any janitor to give you the solution...

Warning Protocol

• Hierarchically nesting elements (e.g., relation/block/tuple) can be locked with intention locks IS and IX

• Idea
  - start locking at the root (relation) level
  - to place an S or X lock on a subelement, first place a corresponding intention lock IS or IX the element itself
  • Warns others: "I'll be reading/writing some subelement of this element"

Example (T₁ reads t₂, T₂ reads R₁)

Example (T₁ reads t₂, T₂ writes to t₄)

Compatibility of multiple granularity locks

Parent locked in | Child can be locked in
---|---
IS | IS, S
IX | IS, S, IX, X
S | [S, IS, not necessary]
X | none

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>X</td>
</tr>
<tr>
<td>IX</td>
<td>X</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
**Rules**

1. Follow multiple granularity comp function
2. Lock root of tree first, any mode
3. Node Q can be locked by Ti in S or IS only if parent(Q) can be locked by Ti in IX or IS
4. Node Q can be locked by Ti in X, IX only if parent(Q) locked by Ti in IX
5. Ti is two-phase
6. Ti can unlock node Q only if none of Q’s children are locked by Ti

**Exercise:**

- Can T2 write element f2.2? What locks will T2 get?

- Can T2 write element f3.1? What locks will T2 get?

- Can T2 read element f2.2? What locks will T2 get?

- Can T2 write element f2.2? What locks will T2 get?
Deletions similar to writes:
- Get an exclusive lock on A before deleting A

Insertions more problematic:
- Possible to lock only existing elements

Phantom tuples:
- Tuples that should have been locked, but did not exist when the locks were taken

Example: relation R (E#, name, ...)
Constraint: E# is key

Use tuple locking

R
<table>
<thead>
<tr>
<th>E#</th>
<th>Name</th>
<th>FInit</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>55</td>
<td>Bush</td>
</tr>
<tr>
<td>t2</td>
<td>75</td>
<td>Clinton</td>
</tr>
</tbody>
</table>

Solution
- Use multiple granularity tree
- Before insert of node Q, lock parent(Q) in X mode

Validation
- Lock-based concurrency control is pessimistic: non-serializable schedules are prevented in advance
- Another, optimistic strategy:
  - Allow transaction Ti access data without locks, but record elements read or written by Ti (in read and write sets RS(Ti) and WS(Ti))
  - At the end, validate that the actions correspond to some serial schedule
Validation
Transactions have 3 phases:
(1) **Read**
   - all accessed DB elements read
   - writes to temporary storage (no locking)
(2) **Validate**
   - check if schedule so far is serializable;
     if yes, then ...
(3) **Write**
   - write updated elements to DB

Key idea
• Make validation an atomic operation
  - i.e., validate a single transaction at a time
• If T₁, T₂, T₃, ... is validation order, then
  resulting schedule will be conflict equivalent to Sₛ = T₁ T₂ T₃...

To implement validation, system maintains two sets of transactions:
• **FIN** = transactions that have finished phase 3 (writing, and are completed)
• **VAL** = transactions that have successfully finished phase 2
  (validation), but not yet completed

Example of what validation must prevent:
\[ RS(T₂) = \{B\} \quad \cap \quad RS(T₃) = \{A, B\} \]
\[ WS(T₂) = \{B, D\} \quad \cap \quad WS(T₃) = \{C\} \]

Another thing validation must prevent:
\[ RS(T₂) = \{A\} \quad \cap \quad RS(T₃) = \{A, B\} \]
\[ WS(T₂) = \{D, E\} \quad \cap \quad WS(T₃) = \{C, D\} \]
Another thing validation must prevent:
RS(T2)={A}    RS(T3)={A,B}
WS(T2)={D,E} ¬ ∩ ¬ WS(T3)={C, D} ≠ ∅

Validation rules for T_j:
(1) When T_j starts phase 1 (reading DB):
   Ignore(T_j) ← FIN; // Transactions that
   // do not affect the validation of T_j
(2) At Validation of T_j:
   if Validates(T_j) then
     VAL ← VAL U {T_j};
     do the write phase;
     FIN ← FIN U {T_j};
     VAL ← VAL - {T_j};
   end if

Validates(T_j): // returns True if T_j validates
for each U ∈ VAL do
  if (WS(U) ∩ RS(T_j) ≠ ∅ or
      WS(U) ∩ WS(T_j) ≠ ∅) then
    return False;
  end if
end for
for each U ∈ FIN - Ignore(T_j) do
  if (WS(U) ∩ RS(T_j) ≠ ∅) then
    return False;
  end if
end for
return True;

Validation is useful in some cases:
- If interaction among transactions low
  → rollbacks rare
- If system resources are plentiful
  - slightly more bookkeeping than for locking
- If there are real-time constraints
  - causes no delays for transactions

Summary
Have studied C.C. mechanisms used in practice
- 2 PL
- Multiple granularity
- Validation