Transactions and Failure Recovery

Instructor: Matei Zaharia

cs245.stanford.edu
Outline

Recap from last time

Undo/redo logging

External actions

Media failures
Outline

Recap from last time

Undo/redo logging

External actions

Media failures
Defining Correctness

**Constraint**: Boolean predicate about DB state (both logical & physical data structures)

**Consistent DB**: satisfies all constraints
Transaction: Collection of Actions that Preserve Consistency

Consistent DB \( T \) Consistent DB’
Our Failure Model

Fail-stop failures of CPU & memory, but disk survives
Undo Logging  (Immediate modification)

T1: Read (A,t);  t ← t×2   A=B
   Write (A,t);
   Read (B,t);  t ← t×2
   Write (B,t);
   Output (A);
   Output (B);

A:8  B:8
memory

A:8  B:8
disk

log
Undo Logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
    Write (A,t);
    Read (B,t); t ← t×2
    Write (B,t);
    Output (A);
    Output (B);

memory

A:8 16
B:8 16

disk

A:8
B:8

log

<T1, start>
<T1, A, 8>
**Undo Logging**  (Immediate modification)

T1: Read (A,t);  \( t \leftarrow t \times 2 \)  \( A=B \)
Write (A,t);
Read (B,t);  \( t \leftarrow t \times 2 \)
Write (B,t);
Output (A);
Output (B);

- Memory:
  - A: 8
  - B: 8

- Disk:
  - A: 16
  - B: 8

- Log:
  - `<T1, start>`
  - `<T1, A, 8>`
  - `<T1, B, 8>`
Undo Logging (Immediate modification)

T1: Read (A,t);  t ← t×2   A=B
  Write (A,t);
  Read (B,t);  t ← t×2
  Write (B,t);
  Output (A);
  Output (B);

memory

A: 8  16
B: 8  16

disk

A: 8  16
B: 8  16

log

<T1, start>
<T1, A, 8>
<T1, B, 8>
Undo Logging  (Immediate modification)

T1: Read (A,t);  t ← t×2       A=B
    Write (A,t);
    Read (B,t);  t ← t×2
    Write (B,t);
    Output (A);
    Output (B);

memory
A:8  16
B:8  16

disk
A:8  16
B:8  16

log
<T1, start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>
Redo Logging  (deferred modification)

T1: Read(A,t); t ← t×2; write (A,t);
    Read(B,t); t ← t×2; write (B,t);
    Output(A); Output(B)
Redo Logging (deferred modification)

T1: Read(A, t); t ← t×2; write (A, t);
    Read(B, t); t ← t×2; write (B, t);
    Output(A); Output(B)
Redo Logging (deferred modification)

T1: Read(A, t); t ← t×2; write (A, t);
Read(B, t); t ← t×2; write (B, t);
Output(A); Output(B)
**Redo Logging** (deferred modification)

T1: Read(A,t); t ← t×2; write (A,t); Read(B,t); t ← t×2; write (B,t); Output(A); Output(B)

A: 8
B: 8

memory

Output

A: 8
16

B: 8
16

DB

A: 8
B: 8

LOG

<T1, start>
<T1, A, 16>
<T1, B, 16>
<T1, commit>
<T1, end>
Combining <Ti, end> Records

Want to delay DB flushes for hot objects

Say X is branch balance:

T1: ... update X...
T2: ... update X...
T3: ... update X...
T4: ... update X...

Actions:
write X
output X
write X
output X
write X
output X
write X
output X
Combining \(<Ti, \text{end}>\) Records

Want to delay DB flushes for hot objects

Say X is branch balance:

T1: ... update X...
T2: ... update X...
T3: ... update X...
T4: ... update X...

Actions:
write X
output X
write X
output X
write X
output X
write X
output X

combined \(<\text{end}>\) record (checkpoint)
**Redo Logging: What To Do at Recovery?**

Redo log (disk):

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;\text{T1},16&gt;</td>
<td>...</td>
<td>&lt;\text{T1},\text{commit}&gt;</td>
<td>...</td>
<td>&lt;\text{checkpoint}&gt;</td>
<td>...</td>
<td>&lt;\text{T2},17&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;\text{T2},\text{commit}&gt;</td>
<td>...</td>
<td>&lt;\text{T2},\text{commit}&gt;</td>
<td>...</td>
<td>&lt;\text{T3},21&gt;</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Redo Logging: What To Do at Recovery?

Redo log (disk):

T2 committed, so REDO all its updates
Redo Logging: What To Do at Recovery?

Redo log (disk):

<T1,A,16>
<T1,commit>
<checkpoint>
<T2,B,17>
<T2,commit>
<T3,C,21>

Crash

T2 committed, so REDO all its updates
T3 didn’t commit, so ignore it
Problems with Ideas So Far

**Undo logging**: need to wait for lots of I/O to commit; can’t easily have backup copies of DB

**Redo logging**: need to keep all modified blocks in memory until commit
Solution: Undo/Redo Logging!

Update = <Ti, X, new X val, old X val>

(X is the object updated)
Undo/Redo Logging Rules

Object X can be flushed before or after Ti commits

Log record (with undo/redo info) must be flushed before corresponding data (WAL)

Flush only commit record at Ti commit
# Undo/Redo Logging: What to Do at Recovery?

## Undo/redo log (disk):

<table>
<thead>
<tr>
<th>...</th>
<th>&lt;checkpoint&gt;</th>
<th>...</th>
<th>&lt;T1, A, 10, 15&gt;</th>
<th>...</th>
<th>&lt;T1, B, 20, 23&gt;</th>
<th>...</th>
<th>&lt;T1, commit&gt;</th>
<th>...</th>
<th>&lt;T2, C, 30, 38&gt;</th>
<th>...</th>
<th>&lt;T2, D, 40, 41&gt;</th>
<th>...</th>
<th>Crash</th>
</tr>
</thead>
</table>

Undo/Redo Logging: What to Do at Recovery?

Undo/redo log (disk):

| ... | <checkpoint> | ... | <T1, A, 10, 15> | ... | <T1, B, 20, 23> | ... | <T1, commit> | ... | <T2, C, 30, 38> | ... | <T2, D, 40, 41> | Crash |

T1 committed, so REDO all its updates
Undo/Redo Logging: What to Do at Recovery?

Undo/redo log (disk):

| ... | <checkpoint> | ... | <T1, A, 10, 15> | ... | <T1, B, 20, 23> | ... | <T1, commit> | ... | <T2, C, 30, 38> | ... | <T2, D, 40, 41> | ... | Crash |

- T1 committed, so REDO all its updates
- T2 didn’t commit, so UNDO all its updates
Non-Quiescent Checkpoints

LOG

Start-ckpt
active txs: T1,T2,...

end ckpt

for dirty memory
undo pages flushed
Non-Quiescent Checkpoints

checkpoint process:

for i := 1 to M do
    Output(buffer i)

[transactions run concurrently]
Example 1: How to Recover?

LOG

| ... | T1,-a | ... | Ckpt T1 | ... | Ckpt end | ... | T1,-b |

no T1 commit
Example 1: How to Recover?

```
T1,-a
... Ckpt T1 ...
... Ckpt end ...
T1,-b
```

no T1 commit

Undo T1 (undo a,b)
Example 2: How to Recover?

<table>
<thead>
<tr>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ckpt-s</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>ckpt-end</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>cmt</td>
</tr>
</tbody>
</table>
Example 2: How to Recover?

Redo T1 (redo b,c)
What if a Checkpoint Did Not Complete?

<table>
<thead>
<tr>
<th>L O G</th>
<th>ckpt start</th>
<th>ckpt end</th>
<th>T1 b</th>
<th>ckpt-start</th>
<th>T1 c</th>
<th>...</th>
</tr>
</thead>
</table>

start of last complete checkpoint

Start recovery from last **complete** checkpoint
Undo/Redo Recovery Algorithm

Backward pass (end of log → latest valid checkpoint start)
  » construct set $S$ of committed transactions
  » undo actions of transactions not in $S$

Undo pending transactions
  » follow undo chains for transactions in
    (checkpoint’s active list) - $S$

Forward pass (latest checkpoint start → end of log)
  » redo actions of all transactions in $S$
Outline

Recap from last time

Undo/redo logging

External actions

Media failures
External Actions

E.g., dispense cash at ATM

\[ T_i = a_1 \ a_2 \ldots \ a_j \ldots \ a_n \]
Solution

(1) Execute real-world actions after commit

(2) Try to make idempotent
Solution

(1) Execute real-world actions after commit

(2) Try to make idempotent

ATM

Give $$
\text{give}(\text{amt})$

$$(\text{amt}, \text{Tid}, \text{time})$$
How Would You Handle These Other External Actions?

Charge a customer’s credit card

Cancel someone’s hotel room

Send data into a streaming system
Outline

Recap from last time

Undo/redo logging

External actions

Media failures
Media Failure
(Loss of Nonvolatile Storage)

A: 16
Media Failure
(Loss of Nonvolatile Storage)

Solution: Make copies of data!
Example 1: 3-Way Redundancy

Keep 3 copies on separate disks

Output(X) → three outputs

Input(X) → three inputs + vote
Example 2: Redundant Writes, Single Reads

Keep N copies on separate disks

Output(X) → N outputs

Input(X) → Input one copy
- if ok, done; else try another one

Assumes bad data can be detected!
Example 3: DB Dump + Log

If active database is lost,
   – restore active database from backup
   – bring up-to-date using redo entries in log
Backup Database

Just like a checkpoint, except that we write the full database

create backup database:

for i := 1 to DB_Size do
    [read DB block i; write to backup]

[transactions run concurrently]
Backup Database

Just like a checkpoint, except that we write the full database

create backup database:

for i := 1 to DB_Size do
    [read DB block i; write to backup]

[transactions run concurrently]

Restore from backup DB and log:
Similar to recovery from checkpoint and log
When Can Logs Be Discarded?

- Log
- Last needed undo
- DB dump
- Last needed undo
- Check-point

- Not needed for media recovery
- Not needed for media recovery redo
- Not needed for undo after system failure
- Not needed for redo after system failure

Time
Summary

Consistency of data: maintain constraints

One source of problems: failures
  » Logging
  » Redundancy

Another source of problems: data sharing
  » We’ll cover this next!
Concurrency Control

Instructor: Matei Zaharia

cs245.stanford.edu
The Problem

Different transactions may need to access data items at the same time, violating constraints.
Example

Constraint: all interns have equal salaries

**T1:** add $1000 to each intern’s salary

**T2:** double each intern’s salary

Salaries:

<table>
<thead>
<tr>
<th>Salary</th>
<th>Original</th>
<th>New (T1)</th>
<th>New (T2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>3000</td>
<td>4000</td>
<td>4000</td>
</tr>
<tr>
<td>3000</td>
<td>3000</td>
<td>4000</td>
<td>4000</td>
</tr>
<tr>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td>5000</td>
</tr>
<tr>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td></td>
</tr>
</tbody>
</table>
The Problem

Even if each transaction maintains constraints by itself, interleaving their actions does not

Could try to run just one transaction at a time (serial schedule), but this has problems
  » Too slow! Especially with external clients & IO
High-Level Approach

Define *isolation levels*: sets of guarantees about what transactions may experience

Strongest level: *serializability* (result is same as some serial schedule)

Many others possible: *snapshot isolation*, *read committed*, *read uncommitted*, …
Fundamental Tradeoff

Weaker isolation level

Stronger isolation level

Easier to reason about (can’t see others’ changes)

See others’ changes, but more concurrency
Interesting Fact

SQL standard defines serializability as “same as a serial schedule”, but then also lists 3 types of “anomalies” to define levels:

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Reads</th>
<th>Unrepeatable Reads</th>
<th>Phantom Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Read committed</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Serializable</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Interesting Fact

There are isolation levels other than serializability that meet the last definition!
   » I.e. don’t exhibit those 3 anomalies

Virtually no commercial DBs do serializability by default, and some can’t do it at all

Time to call the lawyers?
In This Course

We’ll first discuss how to offer serializability
  » Many ideas apply to other isolation levels

We’ll see other isolation levels after
Outline

What makes a schedule serializable?
Conflict serializability
Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
Example

T1: Read(A)
   A ← A + 100
   Write(A)
   Read(B)
   B ← B + 100
   Write(B)

T2: Read(A)
   A ← A × 2
   Write(A)
   Read(B)
   B ← B × 2
   Write(B)

Constraint: A = B
Schedule A

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

CS 245
## Schedule B

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

T1

- Read(A); A ← A+100
- Write(A);
- Read(B); B ← B+100;
- Write(B);
### Schedule B

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read(A); A ← A+100</strong>&lt;br&gt;Write(A);</td>
<td><strong>Read(A); A ← A×2;</strong>&lt;br&gt;Write(A);</td>
</tr>
<tr>
<td><strong>Read(B); B ← B+100;</strong>&lt;br&gt;Write(B);</td>
<td><strong>Read(B); B ← B×2;</strong>&lt;br&gt;Write(B);</td>
</tr>
</tbody>
</table>
## Schedule C

<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>Read(A); A ← A×2;</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td>Write(B);</td>
<td>250</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
## Schedule D

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

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Schedule E</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>------------</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A+100</td>
<td>Read(A); A ← A+50; Read(B); B ← B+50; Write(B);</td>
</tr>
<tr>
<td></td>
<td>Write(A);</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>Write(B);</td>
<td></td>
</tr>
</tbody>
</table>
## Schedule E

Same as Schedule D but with new T2’

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>175</td>
<td>75</td>
</tr>
<tr>
<td>175</td>
<td>175</td>
<td></td>
</tr>
</tbody>
</table>
Our Goal

Want schedules that are “good”, regardless of
» initial state and
» transaction semantics

Only look at order of read & write operations

Example:

$S_C = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$
Example:

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

\[ S_C' = r_1(A)w_1(A) r_1(B)w_1(B) r_2(A)w_2(A) r_2(B)w_2(B) \]
However, for $S_D$:

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

Another way to view this:

- $r_1(B)$ after $w_2(B)$ means $T_1$ should be after $T_2$ in an equivalent serial schedule ($T_2 \rightarrow T_1$)
- $r_2(A)$ after $w_1(A)$ means $T_2$ should be after $T_1$ in an equivalent serial schedule ($T_1 \rightarrow T_2$)
- Can’t have both of these!
Outline

What makes a schedule serializable?

- Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  - Shared and exclusive locks
  - Lock tables and multi-level locking

Optimistic concurrency with validation
Concepts

Transaction: sequence of $r_i(x)$, $w_i(x)$ actions

Conflicting actions: $r_1(A)$ $w_1(A)$ $w_1(A)$ $w_2(A)$ $r_2(A)$ $w_2(A)$

Schedule: a chronological order in which all the transactions’ actions are executed

Serial schedule: no interleaving of actions from different transactions
Question

Is it OK to model reads & writes as occurring at a single point in time in a schedule?

\[ S = \ldots \ r_1(x) \ \ldots \ w_2(b) \ \ldots \]
Question

What about conflicting, concurrent actions on same object?

Assume “atomic actions” that only occur at one point in time (e.g. implement using locking)
Definition

$S_1, S_2$ are **conflict equivalent** schedules if $S_1$ can be transformed into $S_2$ by a series of **swaps** of non-conflicting actions

(i.e., can reorder non-conflicting operations in $S_1$ to obtain $S_2$)
Definition

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

Key idea:
» Conflicts “change” result of reads and writes
» Conflict serializable means there exists some equivalent serial execution that does not change the effects.

How can we compute whether a schedule is conflict serializable?
Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
Precedence Graph $P(S)$

Nodes: transactions in a schedule $S$

Edges: $T_i \rightarrow T_j$ whenever

» $p_i(A)$, $q_j(A)$ are actions in $S$

» $p_i(A) <_S q_j(A)$ (occurs earlier in schedule)

» at least one of $p_i$, $q_j$ is a write (i.e. conflict)
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?
Another Exercise

What is $P(S)$ for

$S = w_1(A) \, r_2(A) \, r_3(A) \, w_4(A)$
Lemma

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

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

Proof:

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

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

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

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

$\begin{cases} p_i, q_j \text{ conflict} \\ \end{cases}$

$\Rightarrow S_1, S_2$ not conflict equivalent
Note: \( P(S_1) = P(S_2) \not\Rightarrow S_1, S_2 \text{ conflict equivalent} \)
**Note:** \( P(S_1) = P(S_2) \not\Rightarrow S_1, S_2 \) conflict equivalent

**Counter example:**

\[
S_1 = w_1(A) \ r_2(A) \ w_2(B) \ r_1(B)
\]

\[
S_2 = r_2(A) \ w_1(A) \ r_1(B) \ w_2(B)
\]
Theorem

\[ P(S_1) \text{ acyclic } \iff S_1 \text{ conflict serializable} \]

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

\[ \Rightarrow \exists S_s \text{ (serial): } S_s, S_1 \text{ conflict equivalent} \]

\[ \Rightarrow P(S_s) = P(S_1) \text{ (by previous lemma)} \]

\[ \Rightarrow P(S_1) \text{ acyclic since } P(S_s) \text{ is acyclic} \]
Theorem

\[ P(S_1) \text{ acyclic } \iff S_1 \text{ conflict serializable} \]
Theorem

$P(S_1)$ 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 inbound edges
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!