Concurrency control & recovery system
Transaction Processing (optional)

Introduction to Database Management
CS348 Fall 2022
Review

• ACID
  • Atomicity: TX’s are either completely done or not done at all
  • Consistency: TX’s should leave the database in a consistent state
  • Isolation: TX’s must behave as if they are executed in isolation
  • Durability: Effects of committed TX’s are resilient against failures

• SQL transactions
  -- Begins implicitly
  SELECT ...;
  UPDATE ...;
  ROLLBACK | COMMIT;
Outline

• Concurrency control -- isolation
  • Review serializable execution histories
  • Locking-based concurrency control

• Recovery – atomicity and durability
  • Naïve approaches
  • Logging for undo and redo
Concurrency control

• Goal: ensure the “I” (isolation) in ACID

$T_1$:  
r1(x);
w1(x);
r1(y);
w1(y);
commit;

$T_2$:  
r2(x);
w2(x);
r2(z);
w2(z);
commit;
## Good versus bad execution histories

### Serial

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1(x)$</td>
<td>$w_1(x)$</td>
</tr>
<tr>
<td>$r_1(y)$</td>
<td>$w_1(y)$</td>
</tr>
<tr>
<td>$r_2(x)$</td>
<td>$w_2(x)$</td>
</tr>
<tr>
<td>$r_2(z)$</td>
<td>$w_2(z)$</td>
</tr>
</tbody>
</table>

$H_a$

### Good!

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1(x)$</td>
<td>$w_1(x)$</td>
</tr>
<tr>
<td>$r_1(y)$</td>
<td>$w_1(y)$</td>
</tr>
</tbody>
</table>

Good! Why?

### Bad!

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1(x)$</td>
<td>$w_1(x)$</td>
</tr>
<tr>
<td>$r_2(x)$</td>
<td>$w_2(x)$</td>
</tr>
<tr>
<td>$r_1(y)$</td>
<td>$w_1(y)$</td>
</tr>
</tbody>
</table>

Read 400

Write 400 – 100

### Good! Why?

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1(x)$</td>
<td>$w_1(x)$</td>
</tr>
<tr>
<td>$r_2(x)$</td>
<td>$w_2(x)$</td>
</tr>
<tr>
<td>$r_1(y)$</td>
<td>$w_1(y)$</td>
</tr>
</tbody>
</table>

Read 400

Write 400 – 50

### Good!

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1(x)$</td>
<td>$w_1(x)$</td>
</tr>
<tr>
<td>$r_2(x)$</td>
<td>$w_2(x)$</td>
</tr>
<tr>
<td>$r_2(C)$</td>
<td>$w_1(y)$</td>
</tr>
</tbody>
</table>

$H_c$
Good versus bad execution histories

Serialization graph (Lecture 17)
Good versus bad execution histories

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1(x)</td>
<td></td>
<td>r2(x)</td>
</tr>
<tr>
<td>w1(x)</td>
<td></td>
<td>w2(x)</td>
</tr>
<tr>
<td>r1(y)</td>
<td></td>
<td>r2(z)</td>
</tr>
<tr>
<td>w1(y)</td>
<td></td>
<td>w2(z)</td>
</tr>
</tbody>
</table>

Not serializable

Bad!

How to avoid this?
Concurrency control

Possible classification

• Pessimistic – assume that conflicts will happen and take preventive action
  • Two-phase locking (2PL)
  • Timestamp ordering

• Optimistic – assume that conflicts are rare and run transactions and fix if there is a problem

• We will only review 2PL
Locking

• Rules
  • If a transaction wants to read an object, it must first request a **shared lock (S mode)** on that object
  • If a transaction wants to modify an object, it must first request an **exclusive lock (X mode)** on that object
  • Allow one exclusive lock, or multiple shared locks

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Mode of the lock requested</th>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Compatibility matrix
Basic locking is not enough

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lock-X(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r1(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>w1(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r2(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>w2(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock(x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r2(y)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>w2(y)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock(y)</td>
<td></td>
</tr>
</tbody>
</table>

Possible schedule under locking

But still not conflict-serializable!
Basic locking is not enough

Add 1 to both $x$ and $y$ (preserve $x=y$)

Possible schedule under locking

But still not conflict-serializable!

Multiply both $x$ and $y$ by 2 (preserves $x=y$)

---

Possible schedule under locking:

$T_1$:
- Read 100
- Write 100+1
- Lock-X($x$)
- Read 100
- Write 100*2
- Lock-X($y$)
- Read 200
- Write 200+1
- Lock-X($y$)
- Read 101
- Write 101*2
- Unlock($x$)
- Unlock($y$)

$T_2$:
- Lock-X($x$)
- Read 101
- Write 101*2
- Unlock($x$)
- Lock-X($y$)
- Read 100
- Write 100*2
- Unlock($y$)

$x \neq y$!
Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X($x$)</td>
<td>r1($x$)</td>
</tr>
<tr>
<td>r1($x$)</td>
<td>w1($x$)</td>
</tr>
<tr>
<td>lock-X($y$)</td>
<td></td>
</tr>
<tr>
<td>unlock($x$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>r1($y$)</td>
<td>w1($y$)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>lock-X($y$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>r2($y$)</td>
<td>w2($y$)</td>
</tr>
<tr>
<td>unlock($y$)</td>
<td></td>
</tr>
</tbody>
</table>

2PL guarantees a conflict-serializable schedule

Cannot obtain the lock on $y$ until $T_1$ unlocks
### Remaining problems of 2PL

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1(x)</td>
<td>r2(x)</td>
</tr>
<tr>
<td>w1(x)</td>
<td>w2(x)</td>
</tr>
<tr>
<td>r1(y)</td>
<td>r2(y)</td>
</tr>
<tr>
<td>w1(y)</td>
<td>w2(y)</td>
</tr>
</tbody>
</table>

- $T_2$ has read uncommitted data written by $T_1$
- If $T_1$ aborts, then $T_2$ must abort as well
- **Cascading aborts** possible if other transactions have read data written by $T_2$

- Even worse, what if $T_2$ commits before $T_1$?
  - Schedule is **not recoverable** if the system crashes right after $T_2$ commits
Deadlocks

• A transaction is deadlocked if it is blocked and will remain blocked until there is an intervention.

• Locking-based concurrency control algorithms may cause deadlocks requiring abort of one of the transactions.

• Consider the partial history
  • Neither $T_1$ nor $T_2$ can make progress

  Cannot obtain the lock on $y$ until $T_2$ unlocks

  Cannot obtain the lock on $y$ until $T_1$ unlocks
Strict 2PL

- Only release X-locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts

- Used in many commercial DBMS
  - Oracle is a notable exception

- Why do we use strict 2PL? (assignment question)
Outline

• Concurrency control -- isolation
  • Review serializable execution histories
  • Locking-based concurrency control

• Recovery – atomicity and durability
  • Naïve approaches
  • Logging for undo and redo
Execution model

To read/write X

• The disk block containing X must be first brought into memory
• X is read/written in memory
• The memory block containing X, if modified, must be written back (flushed) to disk eventually
Failures

• System crashes right after a transaction $T_1$ commits; but not all effects of $T_1$ were written to disk
  • How do we complete/redo $T_1$ (durability)?

• System crashes in the middle of a transaction $T_2$; partial effects of $T_2$ were written to disk
  • How do we undo $T_2$ (atomicity)?
Naïve approach: Force -- durability

**T1** (balance transfer of $100 from A to B)

```
read(A, a); a = a – 100;
write(A, a);
read(B, b); b = b + 100;
write(B, b);
commit;
```

**Force:** all writes must be reflected on disk when a transaction commits
Naïve approach: Force -- durability

\( T1 \) (balance transfer of $100 from A to B)

read(A, a); \( a = a - 100 \);
write(A, a);
read(B, b); \( b = b + 100 \);
write(B, b);
commit;

Force: all writes must be reflected on disk when a transaction commits

Without force: not all writes are on disk when T1 commits
If system crashes right after \( T1 \) commits, effects of \( T1 \) will be lost
**Naïve approach: No steal -- atomicity**

**T1** (balance transfer of $100 from A to B)

read(A, a); $a = a - 100$;
write(A, a);
read(B, b); $b = b + 100$;
write(B, b);
commit;

**No steal:** Writes of a transaction can only be flushed to disk at commit time:
- e.g. $A=700$ cannot be flushed to disk before commit.

With steal: some writes are on disk before T commits

If system crashes before T1 commits, there is no way to undo the changes

---

<table>
<thead>
<tr>
<th>Memory buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 800 700</td>
</tr>
<tr>
<td>B = 400 500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 800 700</td>
</tr>
<tr>
<td>B = 400 500</td>
</tr>
</tbody>
</table>

Bad!
Naïve approach

• **Force**: When a transaction commits, all writes of this transaction must be reflected on disk
  • Ensures durability
  ☺ Problem of force: Lots of *random writes* hurt performance

• **No steal**: Writes of a transaction can only be flushed to disk at commit time
  • Ensures atomicity
  ☺ Problem of no steal: Holding on to all dirty blocks requires lots of memory
Logging

- **Database log**: sequence of log records, recording all changes made to the database, written to stable storage (e.g., disk) during normal operation.

- Hey, one change turns into two—bad for performance?
  - But writes are **sequential** (append to the end of log).
Log format

• When a transaction $T_i$ starts
  • $\langle T_i, \text{start} \rangle$

• Record values before and after each modification:
  • $\langle T_i, X, \text{old_value_of}_X, \text{new_value_of}_X \rangle$
    • $T_i$ is transaction id
    • $X$ identifies the data item

• A transaction $T_i$ is committed when its commit log record is written to disk
  • $\langle T_i, \text{commit} \rangle$
When to write log records into stable store?

• **Write-ahead logging (WAL):** Before \( X \) is modified on disk, the log record pertaining to \( X \) must be flushed.

• Without WAL, system might crash after \( X \) is modified on disk but before its log record is written to disk—no way to undo.
**Undo/redo logging example**

*T1* (balance transfer of $100 from *A* to *B*)

read(*A*, *a*); \( a = a - 100 \);
write(*A*, *a*);
read(*B*, *b*); \( b = b + 100 \);
write(*B*, *b*);

```
Memory buffer
\[
\begin{array}{c}
A = 800 \\
B = 400 \\
\end{array}
\]
```

```
Disk
\[
\begin{array}{c}
A = 800 \\
B = 400 \\
\end{array}
\]
```

```
Log
\[
\langle T_1, \text{start} \rangle \\
\langle T_1, A, 800, 700 \rangle \\
\langle T_1, B, 400, 500 \rangle \\
\]
```

WAL: Before *A*,*B* are modified on disk, their log info must be flushed
Undo/redo logging example cont.

**T1** (balance transfer of $100 from A to B)

read(A, a); $a = a - 100$;
write(A, a);
read(B, b); $b = b + 100$;
write(B, b);

Memory buffer

<table>
<thead>
<tr>
<th>A</th>
<th>800</th>
<th>700</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>400</td>
<td>500</td>
</tr>
</tbody>
</table>

Steal: can flush before commit

Disk

<table>
<thead>
<tr>
<th>A</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>400</td>
</tr>
</tbody>
</table>

Log

\[
\langle T_1, \text{start} \rangle
\]

\[
\langle T_1, A, 800, 700 \rangle
\]

\[
\langle T_1, B, 400, 500 \rangle
\]

If system crashes before T1 commits, we have the old value of A stored on the log to **undo** T1
Undo/redo logging example cont.

**T1** (balance transfer of $100 from A to B)

read(A, a); $a = a - 100$;
write(A, a);
read(B, b); $b = b + 100$;
write(B, b);
commit;

No force: can flush after commit

If system crashes before we flush the changes of A, B to the disk, we have their new committed values on the log to **redo** T1
Log example

• Redo phase:

List of active transactions at crash:
T₁, T₂, T₃

Log

Start of log
redo
t₁, start
redo
t₁, x, 99, 100
redo
t₂, start
redo
t₂, y, 199, 200
redo
t₃, start
redo
t₃, z, 51, 50
redo
t₂, w, 1000, 10
redo
t₂, commit
redo
t₄, start
redo
t₃, abort
redo
t₄, y, 200, 50

End of log
Log example

• Redo phase:

List of active transactions at crash:

T1  T2  T3

End of log

Start of log

Log

T1, start
T1, x, 99, 100
T2, start
T2, y, 199, 200
T3, start
T3, z, 51, 50
T2, w, 1000, 10
T2, commit
T4, start
T3, abort
T4, y, 200, 50
Log example

- Redo phase:

List of active transactions at crash:
T1 T2 T3 T4

- Log example

- Redo phase:

Log

- Start of log
  - redo
  - redo
  - redo
  - redo
  - redo
  - redo
  - redo

- End of log
  - redo
  - redo
  - redo

- Log

  - T1, start
  - T1, x, 99, 100
  - T2, start
  - T2, y, 199, 200
  - T3, start
  - T3, z, 51, 50
  - T2, w, 1000, 10
  - T2, commit
  - T4, start
  - T3, abort
  - T4, y, 200, 50

CRASH!!
Log example

- Redo phase:

List of active transactions at crash: T1, T2, T3, T4

Log

Start of log

$T_1$, start
$T_1$, x, 99, 100
$T_2$, start
$T_2$, y, 199, 200
$T_3$, start
$T_3$, z, 51, 50
$T_2$, w, 1000, 10
$T_2$, commit
$T_4$, start
$T_4$, y, 200, 50

End of log
Log example

• Redo phase:

List of active transactions at crash: T1 T2 T3 T4

End of log

Start of log

Redo phase:

x: 99
y: 199
z: 51
w: 1000

100
200
50
50
10

T1, start
T1, x, 99, 100
T2, start
T2, y, 199, 200
T3, start
T3, z, 51, 50
T2, w, 1000, 10
T2, commit
T4, start
T3, abort
T4, y, 200, 50
Log example

• Undo phase: T1, T4

List of active transactions at crash:
T1, T2, T3, T4
Undo/redo logging

• U: used to track the set of active transactions at crash

• Redo phase: scan forward to end of the log
  • For a log record \( \langle T, \text{start} \rangle \), add \( T \) to \( U \)
  • For a log record \( \langle T, \text{commit} | \text{abort} \rangle \), remove \( T \) from \( U \)
  • For a log record \( \langle T, X, \text{old}, \text{new} \rangle \), issue write\((X, \text{new})\)
    \( \text{Basically repeats history!} \)

• Undo phase: scan log backward
  • Undo the effects of transactions in \( U \)
  • That is, for each log record \( \langle T, X, \text{old}, \text{new} \rangle \) where \( T \) is in \( U \),
  issue write\((X, \text{old})\), and log this operation too (part of the
  “repeating-history” paradigm)
  • Log \( \langle T, \text{abort} \rangle \) when all effects of \( T \) have been undone
Checkpointing

• Shortens the amount of log that need to be undone or redone when a failure occurs

• A checkpoint record contains a list of active transactions

• Steps:
  1. Write a `begin_checkpoint` record into the log
  2. Collect the checkpoint data into the stable storage
  3. Write an `end_checkpoint` record into the log
Summary

• Concurrency control
  • 2PL: guarantees a conflict-serializable schedule
  • Deadlock problem

• Recovery: undo/redo logging
  • Normal operation: write-ahead logging, no force, steal
  • Recovery: first redo (forward), and then undo (backward)