Concurrency control & recovery system
Transaction Processing
Introduction to Database Management
CS348 Spring 2021
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

\[ T_1: \]
\[ r_1(x); \]
\[ w_1(x); \]
\[ r_1(y); \]
\[ w_1(y); \]
\[ \text{commit;} \]

\[ T_2: \]
\[ r_2(x); \]
\[ w_2(x); \]
\[ r_2(z); \]
\[ w_2(z); \]
\[ \text{commit;} \]
## Good versus bad execution histories

### Serial

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

**Good!**

**Read 400**

**Write 400 – 100**

**Bad!**

**Read 400**

**Write 400 – 50**

**Good! Why?**
Good versus bad execution histories

Serialization graph (Lecture 15)
Good versus bad execution histories

How to avoid this?

Not serializable

Bad!

\[ T_1 \]
- \( r_1(x) \)
- \( w_1(x) \)
- \( r_1(y) \)
- \( w_1(y) \)
- \( H_b \)

\[ T_2 \]
- \( r_2(x) \)
- \( w_2(x) \)
- \( r_2(z) \)
- \( w_2(z) \)

\[ T_1 \]

\[ T_2 \]
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>X</td>
<td>X</td>
<td>No</td>
</tr>
</tbody>
</table>

Compatibility matrix
Basic locking is not enough

Possible schedule under locking

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

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

Read 100
Write 100+1

Possible schedule under locking

But still not conflict-serializable!

lock-$X(x)$
$r1(x)$
$w1(x)$
unlock($x$)

lock-$X(x)$
$r2(x)$
$w2(x)$
unlock($x$)

lock-$X(y)$
$r2(y)$
$w2(y)$
unlock($y$)

lock-$X(y)$
$r1(y)$
$w1(y)$
unlock($y$)

lock-$X(y)$
$r2(y)$
$w2(y)$
unlock($y$)

$T_1$

$T_2$

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

Read 101
Write 101*2

Read 100
Write 100*2

$T_1$

$T_2$

$T_1$

$T_2$

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

Read 100
Write 100+1

Possible schedule under locking

But still not conflict-serializable!

lock-$X(x)$
$r1(x)$
$w1(x)$
unlock($x$)

lock-$X(y)$
$r2(y)$
$w2(y)$
unlock($y$)

lock-$X(y)$
$r1(y)$
$w1(y)$
unlock($y$)

lock-$X(y)$
$r2(y)$
$w2(y)$
unlock($y$)

lock-$X(y)$
$r2(y)$
$w2(y)$
unlock($y$)

$T_1$

$T_2$

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

Read 101
Write 101*2

Read 100
Write 100*2

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$

$T_1$

$T_2$
Two-phase locking (2PL)

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

\[
\begin{align*}
T_1 & \quad T_2 \\
\text{lock-X}(x) & \quad \text{lock-X}(x) \\
r_1(x) & \quad r_2(x) \\
w_1(x) & \quad w_2(x) \\
\text{lock-X}(y) & \quad \text{lock-X}(y) \\
\text{unlock}(x) & \quad r_2(y) \\
\text{unlock}(y) & \quad w_2(y)
\end{align*}
\]

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>$r_1(x)$</td>
<td>$r_2(x)$</td>
</tr>
<tr>
<td>$w_1(x)$</td>
<td>$w_2(x)$</td>
</tr>
<tr>
<td>$r_1(y)$</td>
<td>$r_2(y)$</td>
</tr>
<tr>
<td>$w_1(y)$</td>
<td>$w_2(y)$</td>
</tr>
<tr>
<td>Abort!</td>
<td></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

\[ T1 \text{ (balance transfer of $100 from A to B)} \]

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)

```plaintext
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
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, the 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

\( T_1 \) (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);

WAL: Before A,B are modified on disk, their log info must be flushed
Undo/red 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);

If system crashes before T1 commits, we have the old value of A stored on the log to undo T1

Steal: can flush before commit
Undo/redo logging example cont.

\( T_1 \) (balance transfer of $100 from A to B)

\[
\begin{align*}
\text{read}(A, a); & \quad a = a - 100; \\
\text{write}(A, a); \\
\text{read}(B, b); & \quad b = b + 100; \\
\text{write}(B, b); \\
\text{commit}; \\
\end{align*}
\]

---

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 \( T_1 \)
Log example

• Redo phase:

List of active transactions at crash:
T1  T2  T3

Log

Start of log

redo
redo
redo
redo
redo
redo
redo
redo

End of 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:
  - Log

  \begin{align*}
  x & : 99 \quad 100 \\
  y & : 199 \quad 200 \\
  z & : 51 \quad 50 \\
  w & : 1000 \quad 10 \\
  \end{align*}

  List of active transactions at crash:
  \( T_1 \ T_2 \ T_3 \)

  

log

\textbf{Start of log}

\begin{align*}
T_1, \text{ start} \\
T_1, x, 99, 100 \\
T_2, \text{ start} \\
T_2, y, 199, 200 \\
T_3, \text{ start} \\
T_3, z, 51, 50 \\
T_2, w, 1000, 10 \\
T_2, \text{ commit} \\
T_4, \text{ start} \\
T_3, \text{ abort} \\
T_4, y, 200, 50 \\
\end{align*}

End of log

\textbf{End of log}
Log example

- Redo phase:

List of active transactions at crash:
T1 T2 T3 T4

\[
\begin{align*}
x &: 99 & 100 \\
y &: 199 & 200 \\
z &: 51 & 50 \\
w &: 1000 & 10
\end{align*}
\]

Start of log

\[
\begin{align*}
T_1, \text{ start} \\
T_1, x, 99, 100 \\
T_2, \text{ start} \\
T_2, y, 199, 200 \\
T_3, \text{ start} \\
T_3, z, 51, 50 \\
T_2, w, 1000, 10 \\
T_2, \text{ commit} \\
T_4, \text{ start} \\
T_3, \text{ abort} \\
T_4, y, 200, 50
\end{align*}
\]

End of log
Log example

- Redo phase:

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

- Redo phase:
  - Start of log
  - Redo
    - \(T_1\), start
    - \(T_1\), \(x\), 99, 100
    - \(T_2\), start
    - \(T_2\), \(y\), 199, 200
    - \(T_3\), start
    - \(T_3\), \(z\), 51, 50
    - \(T_3\), \(w\), 1000, 10
    - \(T_2\), commit
    - \(T_4\), start
    - \(T_4\), \(y\), 200, 50
  - End of log

List of active transactions at crash:
- \(T_1\)
- \(T_2\)
- \(T_3\)
- \(T_4\)
Log example

- Undo phase: T1, T4

List of active transactions at crash: T1 T2 T3 T4

Start of log

* undo

End of log

undo

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

T4, abort
T1, abort
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 \( \langle T, \text{commit} \mid \text{abort} \rangle \), remove \( T \) from \( U \)
  - For a log record \( \langle T, X, \text{old}, \text{new} \rangle \), issue \( \text{write}(X, \text{new}) \)
    - 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 \( \text{write}(X, \text{old}) \), 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)