Transactions and Failure Recovery

Instructor: Matei Zaharia

cs245.stanford.edu
Outline

Recap from last time

Redo logging

Undo/redo logging

External actions

Media failures
Outline

Recap from last time

Redo logging

Undo/redo logging

External actions

Media failures
Defining Correctness

**Constraint:** Boolean predicate about our DB (both logical and physical data structures)

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

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

T1: Read (A,t); \[ t \leftarrow t \times 2 \] \hspace{1cm} 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:8
B:8

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);

```
<table>
<thead>
<tr>
<th></th>
<th>memory</th>
<th>disk</th>
<th>log</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>16</td>
<td>&lt;T1, start&gt;</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>16</td>
<td>&lt;T1, A, 8&gt;</td>
</tr>
</tbody>
</table>
```
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

disk

log

\(<T1, \text{start}>\)
\(<T1, A, 8>\)
\(<T1, B, 8>\)
**Undo Logging** (Immediate modification)

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

memory

disk

log

\(<T₁, \text{start}>\)
\(<T₁, A, 8>\)
\(<T₁, B, 8>\)
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);

\[
\begin{array}{c|c}
\text{memory} & \text{disk} & \text{log} \\
A:8 & 16 & 16 \\
B:8 & 16 & 16 \\
\end{array}
\]
Downsides of Undo Logging

Have to do a lot of I/O to commit (write all updated objects to disk first)

Hard to replicate database to another disk (must push all changes across the network)
Redo Logging

First send Gretel up with no rope, then Hansel goes up safely with rope!
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

A: 8
B: 8

DB

LOG
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)

memory

DB

LOG
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)

memory

DB

LOG

<T1, start>
<T1, A, 16>
<T1, B, 16>
<T1, commit>
<T1, end>
Redo Logging Rules

1. For every action, generate redo log record (containing new value)
2. Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk
3. Flush log at commit
4. Write END record after DB updates flushed to disk
Recovery Rules: Redo Logging

(1) Let $S$ = set of transactions with $<T_i, \text{commit}>$ (and no $<T_i, \text{end}>$) in log

(2) For each $<T_i, X, v>$ in log, in forward order (earliest $\rightarrow$ latest) do:
   - if $T_i \in S$ then
     \[
     \begin{cases} 
     \text{Write}(X, v) \\
     \text{Output}(X) 
     \end{cases}
     \]

(3) For each $T_i \in S$, write $<T_i, \text{end}>$
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

Combining \textless Ti, end\textgreater  Records

Want to delay DB flushes for hot objects

Say X is branch balance:

\begin{itemize}
  \item T1: ... update X...
  \item T2: ... update X...
  \item T3: ... update X...
  \item T4: ... update X...
\end{itemize}

Actions:

- write X
- output X
- write X
- output X
- write X
- output X
- write X
- output X

combined \textless end\textgreater  record
Solution: Checkpoints

Simple, naïve checkpoint algorithm:
1. Stop accepting new transactions
2. Wait until all transactions finish
3. Flush all log records to disk (log)
4. Flush all buffers to disk (DB) (do not discard buffers)
5. Write “checkpoint” record on disk (log)
6. Resume transaction processing
Example: What To Do at Recovery?

Redo log (disk):

<table>
<thead>
<tr>
<th>...</th>
<th>&lt;T1, A, 16&gt;</th>
<th>...</th>
<th>&lt;T1, commit&gt;</th>
<th>...</th>
<th>Checkpoint</th>
<th>...</th>
<th>&lt;T2, B, 17&gt;</th>
<th>...</th>
<th>&lt;T2, commit&gt;</th>
<th>...</th>
<th>&lt;T3, C, 21&gt;</th>
<th>...</th>
<th>Crash</th>
</tr>
</thead>
</table>
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
Example: 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>Crash</th>
</tr>
</thead>
</table>

...
Example: 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
### Example: 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>
</tr>
</thead>
</table>

**Crash**

- **T1 committed, so REDO all its updates**
- **T2 didn’t commit, so UNDO all its updates**

---

CS 245
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?

<table>
<thead>
<tr>
<th>LOG</th>
<th>...</th>
<th>T₁₋a</th>
<th>...</th>
<th>Ckpt</th>
<th>T₁</th>
<th>...</th>
<th>Ckpt</th>
<th>end</th>
<th>...</th>
<th>T₁₋b</th>
</tr>
</thead>
</table>

no T₁ commit
Example 1: How to Recover?

<table>
<thead>
<tr>
<th>LOG</th>
<th>...</th>
<th>T₁,-a</th>
<th>...</th>
<th>Ckpt T₁</th>
<th>...</th>
<th>Ckpt end</th>
<th>...</th>
<th>T₁-b</th>
</tr>
</thead>
</table>

Undo T₁ (undo a,b)

no T₁ commit
Example 2: How to Recover?

LOG

... T1 a ... ckpt-s T1 ... T1 b ... ckpt-end ... T1 c ... T1 cmt ...
Example 2: How to Recover?

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

Start recovery from last complete checkpoint
Undo/Redo Recovery Process

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

Redo logging

Undo/redo logging

External actions

Media failures
External Actions

E.g., dispense cash at ATM

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

$\downarrow$

$\$$
Solution

(1) Execute external actions after commit

(2) Try to make idempotent
Solution

(1) Execute real-world actions after commit

(2) Try to make idempotent

\[
give(amt) \]

\[
\text{lastTid: } \quad \text{time: } \quad \text{give(amt)}
\]

Give $$ \quad (\text{amt}, \text{Tid}, \text{time})$$
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

Redo logging

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: Triple Modular Redundancy

Keep 3 copies on separate disks

Output(X) --> three outputs

Input(X) --> three inputs + vote

\[ \text{X1} \quad \text{X2} \quad \text{X3} \]
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 checkpoint, except that we write 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 checkpoint, except that we write 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 Log Be Discarded?

- 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
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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3000</td>
<td>3000</td>
<td>3000</td>
<td>4000</td>
<td>4000</td>
</tr>
<tr>
<td></td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
<td>5000</td>
<td>5000</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**

- **Stronger isolation level**
  - Easier to reason about (can’t see others’ changes)

- **Weaker isolation level**
  - See others’ changes, but more concurrency
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 2\textsuperscript{nd} 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
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>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td>125</td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td>250</td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td>250</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td>250</td>
</tr>
</tbody>
</table>
## 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>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Schedule B

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A\times2;</td>
<td>Read(A); A ← A\times2;</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B\times2;</td>
<td>Read(B); B ← B\times2;</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Write(B);</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></td>
<td></td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>

CS 245
## Schedule C

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

<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>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Write(B);</td>
<td>Write(B);</td>
<td>250</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>
### Schedule E

<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+50;</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B+50;</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 E

Same as Schedule D but with new T2’

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

<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></td>
</tr>
<tr>
<td>175</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>175</td>
</tr>
<tr>
<td>175</td>
<td>175</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:

\[ Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]
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!
Returning to $S_C$

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

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