SYSTEM FAILURES

Lecture based on [GUW 17 and GUW 11.6]

Slides based on
Notes 08: Failure recovery
for Stanford CS 245, fall 2002
by Hector Garcia–Molina
This lecture

• Logging of **transactions** in order to allow **recovery** in case of a system failure.
  – Undo logging
  – Redo logging
  – Undo/redo logging

• Reliable disk systems (RAID)
Transactions

• **Transactions** are groups of updates to the database.

• We will not consider the possibility of many **concurrent transactions** (read chapter 18–19 for info on that).

• **Basic property**: Transactions are **atomic** (to maintain consistency).

Handling failures during transactions?
Types of system events

Desired events: See product manuals....

Undesired expected events:
System crash
– memory lost
– cpu halts, resets

that’s it!!

Undesired unexpected: Everything else!
Undesired unexpected: Everything else!

Examples:
• Disk data is lost
• Memory lost without CPU halt
• CPU implodes wiping out universe...

We deal only with expected events
Are we not missing something?

Only disregarding small probability events:
Add low level checks + redundancy to increase make model hold with very high probability.

E.g.,
- Replicate disk storage (later today)
- Memory parity
- CPU checks
Operations

• Input (x): block with x → memory
• Output (x): block with x → disk
• Read (x,t): do input(x) if necessary
  \[ t \leftarrow \text{value of } x \text{ in block} \]
• Write (x,t): do input(x) if necessary
  \[ \text{value of } x \text{ in block} \leftarrow t \]
Undo logging

To enable recovery, database systems use **logging** of changes to data.

Arguably, the simplest logging strategy is **undo logging** (due to Hansel and Gretel, 782 AD; improved in 783 AD to durable undo logging)
Undo logging example

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

A:8 16
B:8 16

memory

A:8 16
B:8 16

disk

<T1, start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>

log
One “complication”

- Log is first written in memory
- Not written to disk on every action

```
A: 8 16
B: 8 16

<T1, start>
<T1, A, 8>
<T1, B, 8>
```

DB

Log

BAD STATE

# 1
One “complication”

- Log is first written in memory
- Not written to disk on every action

```
A: 8 16
B: 8 16

<T1,start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>

BAD STATE
# 2

DB
A: 8
B: 16

Log
<T1, start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>
```
Undo logging rules

(1) For every action generate undo log record (containing old value)

(2) Before $x$ is modified on disk, log records pertaining to $x$ must be on disk ("write ahead logging")

(3) Before commit is flushed to log, all writes of transaction must be reflected on disk
Recovery using undo logging

(1) Let $S$ = set of transactions with $<T_i, \text{start}>$ in log, but no $<T_i, \text{commit}>$ (or $<T_i, \text{abort}>$) record in log.

(2) For each $<T_i, X, v>$ in log, in reverse order do:
   - if $T_i \in S$ then
     - write $(X, v)$
     - output $(X)$

(3) For each $T_i \in S$ do
   - write $<T_i, \text{abort}>$ to log
Problem session

What disadvantages could you foresee for undo logging? Consider for example:

• Whether the log file can be used to generate database from a backup.
• Failure during recovery.
Failure during recovery

No problem!

Undo recovery is *idempotent*:
Performing (parts of) it several times produces the same result
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 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
Recovery using redo logging

(1) Let S = set of transactions with <Ti, commit> in log
(2) For each <Ti, X, v> in log, in forward order do:
   - if T ∈ S then
     \[
     \begin{cases} 
     \text{Write}(X, v) \\
     \text{Output}(X) 
     \end{cases}
     \]
Recovery can be very, very SLOW

Redo log:

First Record (1 year ago) --- T1 wrote A,B Committed a year ago --- Last Record

\[\text{Still, Need to redo after crash!!}\]
Checkpoints (simple version)

Periodically:
(1) Do not accept 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 of using a checkpoint

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>Crash</th>
</tr>
</thead>
</table>

...
Undo/redo logging

Some drawbacks:

• *Undo logging* cannot bring backup DB copies up to date

• *Redo logging* needs to keep all modified blocks in memory until commit

Alternative (using more space):

*Undo/redo* log with entries of the form

<\(T_i, X, \text{New } X \text{ val, Old } X \text{ val}\)>
Undo/redo logging rules

• Page X can be flushed before or after Ti commit.
• Log record flushed before corresponding updated page.
• Flush at commit (log only).
Non-quiescent checkpoint

A prize to the student who explains what "end ckpt" is good for!
Recovery with non-quiescent ckpt

Case 1: No T1 commit

|   | T1,−a |   | Ckpt T1 |   | Ckpt end |   | T1−b |

❌ Undo T1 (undo a,b)

Case 2: T1 has committed

|   | T1 a |   | ckpt-s T1 |   | T1 b |   | ckpt-end |   | T1 c |   | T1 cmt |   |

❌ Redo T1: (redo b,c)
undo/redo recovery in general

• Backwards pass (end of log ↳ latest 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 active list and not in S

• Forward pass (latest checkpoint start ↳ end of log)
  – redo actions of S transactions
Media failures
Or: loss of non-volatile storage

General solution: Make copies of data!
Simple example

• Keep 3 copies on separate disks (a simple error-correcting code).
• Output(X) --> three outputs.
• Input(X) --> three inputs + vote.
• Much better codings exist...

\[ X1 \quad X2 \quad X3 \]
Protection against disk crashes

• Disks have their own error-correction mechanism and therefore rarely return data that is wrong.
• However, a disk may crash entirely.
• Dealing with crashed disks is easier than dealing with wrong data. (Why?)
RAID and erasure codes

- **RAID** = "Redundant Array of Independent Disks".
- **RAID Level 1**: 1 redundant disk that keeps the **parity** of data on other disks. Resists one disk failure.
- **RAID Level 6**: Uses several redundant disks and resists two disk failures.
- The theory of **erasure codes** explains how many redundant disks are needed.
Database backups

- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log
When can log be discarded?

- db dump
- last needed undo
- check-point

- not needed for media recovery
- not needed for undo after system failure
- not needed for redo after system failure
Summary

• We can ensure that transactions are atomic, even in the presence of system failures, using logging techniques.
• We saw 3 logging techniques, each with advantages and disadvantages.
• Separate techniques such as RAID ensure reliable non-resilient storage.