Recovery

• First order of business:
  Failure Model
Events — Desired

Undesired — Expected

Unexpected

Our failure model

CPU

processor

memory

M

disk

D
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….
Is this model reasonable?

**Approach:** Add low level checks + redundancy to increase probability model holds

E.g.,
- Replicate disk storage (stable store)
- Memory parity
- CPU checks

---

**Review: The ACID properties**

- **Atomicity:** All actions in the Xact happen, or none happen.
- **Consistency:** If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation:** Execution of one Xact is isolated from that of other Xacts.
- **Durability:** If a Xact commits, its effects persist.

The **Recovery Manager** guarantees Atomicity & Durability.
**Motivation**

- **Atomicity:**
  - Transactions may abort (“Rollback”).

- **Durability:**
  - What if DBMS stops running? (Causes?)

Desired Behavior after system restarts:
- T1, T2 & T3 should be **durable**.
- T4 & T5 should be **aborted** (effects not seen).

---

**Assumptions**

- Concurrency control is in effect.
  - **Strict 2PL**, in particular.

- Updates are happening “in place”.
  - i.e. data is overwritten on (deleted from) the disk.

- A simple scheme to guarantee Atomicity & Durability?
Handling the Buffer Pool

- **Force** every write to disk?
  - Poor response time.
  - But provides durability.
- **Steal** buffer-pool frames from uncommitted Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

More on Steal and Force

- **STEAL** (why enforcing Atomicity is hard)
  - *To steal frame F:* Current page in F (say P) is written to disk; some Xact holds lock on P.
    - What if the Xact with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P).

- **NO FORCE** (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.
Operations:

- Input (x): block with x → memory
- Output (x): block with x → disk
- Read (x,t): do input(x) if necessary
t ← value of x in block
- Write (x,t): do input(x) if necessary
  value of x in block ← t

Key problem  Unfinished transaction

Example  Constraint: A=B
T_1:  A ← A × 2
      B ← B × 2
T1: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

failure!

A: 8 16
B: 8 16

memory
disk

• Need atomicity: execute all actions of a transaction or none at all
One solution: undo logging  (immediate modification)

due to: Hansel and Gretel, 782 AD

• Improved in 784 AD to durable undo logging

(Okay, Ariadne deserves earlier credit)

Basic Idea: Logging

- Record REDO and UNDO information, for every update, in a log.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.

- Log: An ordered list of REDO/UNDO actions
  - Log record contains:
    - \(<XID, pageID, offset, length, old data, new data>\)
  - and additional control info (which we’ll see soon).
Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update before the corresponding data page gets to disk.
  2. Must write all log records for a Xact before commit.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

Exactly how is logging (and recovery!) done?
- We’ll study the ARIES algorithms.

WAL & the Log

- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each data page contains a pageLSN.
  - The LSN of the most recent log record for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN flushed so far.
- WAL: Before a page is written,
  - pageLSN ≤ flushedLSN
Undo logging (Immediate modification)

\[ T_1: \]
- 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 \((B)\);

One “complication”

- Log is first written in memory
- Not written to disk on every action
One “complication”

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

memory

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

A: 6
B: 8

DB

BAD STATE

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: WAL)
3. Before commit is flushed to log, all writes of transaction must be reflected on disk
Recovery rules: Undo logging

- For every Ti with <Ti, start> in log:
  - If <Ti, commit> or <Ti, abort> in log, do nothing
  - Else For all <Ti, X, v> in log:
    - write (X, v)
    - output (X)
    - Write <Ti, abort> to log

IS THIS CORRECT??

Recovery rules: Undo logging

(1) Let S = set of transactions with <Ti, start> in log, but no <Ti, commit> (or <Ti, abort>) record in log
(2) For each <Ti, X, v> in log, in reverse order (latest → earliest) do:
    - if Ti ∈ S then
      - write (X, v)
      - output (X)
(3) For each Ti ∈ S do
    - write <Ti, abort> to log
What if failure during recovery?
No problem! Undo idempotent

Log Records

LogRecord fields:

- Possible log record types:
  - Update
  - Commit
  - Abort
  - End (signifies end of commit or abort)
  - Compensation Log Records (CLRs)
    - for UNDO actions

prevLSN
XID
type
pageID
length
offset
before-image
after-image

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke
Other Log-Related State

- **Transaction Table:**
  - One entry per active Xact.
  - Contains XID, status (running/commited/aborted), and lastLSN.

- **Dirty Page Table:**
  - One entry per dirty page in buffer pool.
  - Contains recLSN -- the LSN of the log record which *first* caused the page to be dirty.

Normal Execution of an Xact

- **Series of reads & writes, followed by commit or abort.**
  - We will assume that write is atomic on disk.
    - In practice, additional details to deal with non-atomic writes.

- **Strict 2PL.**
- **STEAL, NO-FORCE buffer management, with Write-Ahead Logging.**
To discuss:

- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Checkpoints
- Media failures
**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 rules: Redo logging

- For every Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:
    - Write(X, v)
    - Output(X)

IS THIS CORRECT??

Recovery rules: Redo logging

1. Let S = set of transactions with <Ti, commit> in log
2. For each <Ti, X, v> in log, in forward order (earliest → latest) do:
   - if Ti ∈ S then
     - Write(X, v)
     - Output(X)  

Checkpointing

- Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:
  - **begin_checkpoint** record: Indicates when chkpt began.
  - **end_checkpoint** record: Contains current Xact table and dirty page table. This is a ‘fuzzy checkpoint’:
    - Other Xacts continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
    - No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page. (So it’s a good idea to periodically flush dirty pages to disk!)
  - Store LSN of chkpt record in a safe place (master record).

The Big Picture:
What’s Stored Where

- **LOG**
  - LogRecords
    - prevLSN
    - XID
    - type
    - pageID
    - length
    - offset
    - before-image
    - after-image
- **DB**
  - Data pages each with a pageLSN
  - master record
- **RAM**
  - Xact Table
    - lastLSN
    - status
  - Dirty Page Table
    - recLSN
    - flushedLSN
Simple Transaction Abort

- For now, consider an explicit abort of a Xact.
  - No crash involved.
- We want to “play back” the log in reverse order, UNDOing updates.
  - Get lastLSN of Xact from Xact table.
  - Can follow chain of log records backward via the prevLSN field.
  - Before starting UNDO, write an Abort log record.
    - For recovering from crash during UNDO!

Abort, cont.

- To perform UNDO, must have a lock on data!
  - No problem!
- Before restoring old value of a page, write a CLR:
  - You continue logging while you UNDO!!
  - CLR has one extra field: undonextLSN
    - Points to the next LSN to undo (i.e. the prevLSN of the record we’re currently undoing).
  - CLRs never Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- At end of UNDO, write an “end” log record.
Transaction Commit

- Write **commit** record to log.
- All log records up to Xact’s lastLSN are flushed.
  - Guarantees that flushedLSN ≥ lastLSN.
  - Note that log flushes are sequential, synchronous writes to disk.
  - Many log records per log page.
- Commit() returns.
- Write **end** record to log.

Crash Recovery: Big Picture

- Start from a **checkpoint** (found via master record).
- Three phases. Need to:
  - Figure out which Xacts committed since checkpoint, which failed (**Analysis**).
  - REDO all actions.
    - (repeat history)
  - UNDO effects of failed Xacts.

Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
Recovery: The Analysis Phase

- Reconstruct state at checkpoint.
  - via `end_checkpoint` record.
- Scan log forward from checkpoint.
  - End record: Remove Xact from Xact table.
  - Other records: Add Xact to Xact table, set `lastLSN=LSN`, change Xact status on commit.
  - Update record: If P not in Dirty Page Table,
    - Add P to D.P.T., set its `recLSN=LSN`.

Recovery: The REDO Phase

- We `repeat History` to reconstruct state at crash:
  - Reapply all updates (even of aborted Xacts!), redo CLRs.
- Scan forward from log rec containing smallest `recLSN` in D.P.T. For each CLR or update log rec LSNN, REDO the action unless:
  - Affected page is not in the Dirty Page Table, or
  - Affected page is in D.P.T., but has `recLSN > LSN`, or
  - `pageLSN` (in DB) ≥ `LSN`.
- To REDO an action:
  - Reapply logged action.
  - Set `pageLSN` to `LSN`. No additional logging!
Recovery: The UNDO Phase

ToUndo = \{ l | l a lastLSN of a “loser” Xact\}

Repeat:

- Choose largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN == NULL
  - Write an End record for this Xact.
- If this LSN is a CLR, and undonextLSN != NULL
  - Add undonextLSN to ToUndo
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

Example of Recovery

<table>
<thead>
<tr>
<th>Xact Table</th>
<th>lastLSN</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dirty Page Table

<table>
<thead>
<tr>
<th>recLSN</th>
<th>flushedLSN</th>
</tr>
</thead>
</table>

ToUndo

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>45</td>
<td>T1 End</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
</tbody>
</table>

CRASH, RESTART

prevLSNs
Example: Crash During Restart!

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,05</td>
<td>begin_checkpoint, end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 abort</td>
</tr>
<tr>
<td>40,45</td>
<td>CLR: Undo T1 LSN 10, T1 End</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>70</td>
<td>CRASH, RESTART</td>
</tr>
<tr>
<td>80,85</td>
<td>CLR: Undo T3 LSN 50, T3 end</td>
</tr>
<tr>
<td>90</td>
<td>CLR: Undo T2 LSN 20, T2 end</td>
</tr>
</tbody>
</table>

Additional Crash Issues

- What happens if system crashes during Analysis? During REDO?
- How do you limit the amount of work in REDO?
  - Flush asynchronously in the background.
  - Watch “hot spots”!
- How do you limit the amount of work in UNDO?
  - Avoid long-running Xacts.
Summary of Logging/Recovery

- Recovery Manager guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.

Summary, Cont.

- Checkpointing: A quick way to limit the amount of log to scan on recovery.
- Recovery works in 3 phases:
  - Analysis: Forward from checkpoint.
  - Redo: Forward from oldest recLSN.
  - Undo: Backward from end to first LSN of oldest Xact alive at crash.
- Upon Undo, write CLRs.
- Redo “repeats history”: Simplifies the logic!
Recovery is very, very **SLOW**!

Redo log:

First

... ... ...

T1 wrote A,B

Last

Record

Committed a year ago

(1 year ago) crash!!

-->

STILL, Need to redo after

---

**Solution:** Checkpoint  
(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: what to do at recovery?

- Redo log (disk):

| ... | <T1,A,16> | ... | <T1,commit> | ... | Checkpoint | ... | <T2,B,17> | ... | <T2,commit> | ... | <T3,C,21> | Crash |

Key drawbacks:

- *Undo logging*: cannot bring backup DB copies up to date
- *Redo logging*: need to keep all modified blocks in memory until commit
Solution: undo/redo logging!

Update $\Rightarrow$ $<T_i, Xid, \text{New } X \text{ val}, \text{Old } X \text{ val}>$
page $X$

Rules

- Page $X$ can be flushed before or after $T_i$ commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)
Non-quiesce checkpoint

LOG

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

end ckpt ...

for undo dirty buffer pool pages flushed

Examples what to do at recovery time?

LOG

... T1, a ...

... Ckpt T1 ...

... Ckpt end ...

... T1-b ...

Undo T1 (undo a,b)

no T1 commit
Example

LOG

\[
\begin{array}{ccccccc}
... & T1_a & ... & ckpt-s & T1 & ... & ckpt-end & ... & T1_c & ... & T1_cmt & ...
\end{array}
\]

Redo T1: (redo b, c)

Recovery process:

- **Backwards pass** (end of log \(\Rightarrow\) 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) - S
- **Forward pass** (latest checkpoint start \(\Rightarrow\) end of log)
  - redo actions of S transactions

\[\text{start checkpoint} \quad \text{backward pass} \quad \text{forward pass} \]
Real world 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
ATM

Give $(amt, Tid, time)$

A: 16

Media failure (loss of non-volatile 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

Example #2  Redundant writes, Single reads

- Keep N copies on separate disks
- Output(X) --> N outputs
- Input(X) --> Input one copy
  \[
  \begin{cases} 
  \text{- if ok, done} \\
  \text{- else try another one}
  \end{cases}
  \]

\(\Leftrightarrow\) 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

When can log be discarded?

- Not needed for media recovery
- Not needed for undo after system failure
- Not needed for redo after system failure
More on transaction processing

Topics:
- Cascading rollback, recoverable schedule
- Deadlocks
  - Prevention
  - Detection
- View serializability
- Distributed transactions
- Long transactions (nested, compensation)

Concurrency control & recovery

Example:

\[
\begin{array}{c}
T_j \quad T_i \\
\vdots \quad \vdots \\
W_i(A) \quad \vdots \\
\vdots \quad r_i(A) \\
\vdots \quad \text{Commit } T_i \\
\vdots \quad \vdots \\
\text{Abort } T_j
\end{array}
\]

Cascading rollback  (Bad!)
• Schedule is conflict serializable
• $T_j \rightarrow T_i$

• But not recoverable

• Need to make “final’ decision for each transaction:
  – commit decision - system guarantees transaction will or has completed, no matter what
  – abort decision - system guarantees transaction will or has been rolled back (has no effect)
To model this, two new actions:

- $C_i$ - transaction $T_i$ commits
- $A_i$ - transaction $T_i$ aborts

---

Back to example:

<table>
<thead>
<tr>
<th>$T_j$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_j(A)$</td>
<td>$r_i(A)$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$C_i$ $\leftarrow$ can we commit here?
**Definition**

Ti reads from Tj in S (Tj ⇒_S Ti) if

1. \( w_j(A) <_S r_i(A) \)
2. \( a_j <_S r_i(A) \) \((<_S: does\ not\ precede)\)
3. If \( w_j(A) <_S w_k(A) <_S r_i(A) \) then \( a_k <_S r_i(A) \)

---

**Definition**

Schedule S is **recoverable** if whenever \( T_j \Rightarrow_T S T_i \) and \( j \neq i \) and \( C_i \in S \) then \( C_j <_S C_i \)
Note: in transactions, reads and writes precede commit or abort

\[ \begin{align*}
&\text{If } C_i \in T_i, \text{ then } r_i(A) < C_i \\
&\hspace{1em} w_i(A) < C_i \\
&\text{If } A_i \in T_i, \text{ then } r_i(A) < A_i \\
&\hspace{1em} w_i(A) < A_i
\end{align*} \]

- Also, one of $C_i, A_i$ per transaction

How to achieve recoverable schedules?
With 2PL, hold write locks to commit (strict 2PL)

T_j ---- T_i

\[ \vdots \quad \vdots \]

\[ W_j(A) \quad \vdots \]

\[ \vdots \quad \vdots \]

\[ C_j \quad \vdots \]

\[ u_j(A) \quad r_i(A) \]

\[ \vdots \quad \vdots \]

With validation, no change!
• S is recoverable if each transaction *commits* only after all transactions from which it read have committed.

• S avoids cascading rollback if each transaction may *read* only those values written by committed transactions.