Principles of Database Management Systems

7: Crash Recovery

Pekka Kilpeläinen
(after Stanford CS245 slide originals by Hector Garcia-Molina, Jeff Ullman and Jennifer Widom)

PART II of Course

• Major concern so far:
  – How to manipulate database *efficiently*?
• Now shift focus to:
  – How to ensure *correctness* of database manipulation?

Integrity or correctness of data

• Would like data to be “accurate” or “correct” at all times

<table>
<thead>
<tr>
<th>EMP</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>Green</td>
<td>3421</td>
<td></td>
</tr>
<tr>
<td>Gray</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Definition:

• Consistent state: satisfies all constraints
• Consistent DB: DB in a consistent state

Integrity or consistency constraints

• Predicates that the data must satisfy
• Examples:
  - x is key of relation R
  - x → y holds in R
  - Domain(x) = {Red, Blue, Green}
  - no employee should make more than twice the average salary

Constraints (as we use here) may *not* capture “full correctness”

Example 1  Transaction constraints

• When salary is updated, new salary > old salary
• When account record is deleted, balance = 0

In general, various policies to ensure that the DB corresponds to “real world”
Obs: DB cannot be consistent always!
Example: $a_1 + a_2 + \ldots + a_n = \text{TOT}$ (constraint)
Deposit $100$ in $a_2$: $a_2 \leftarrow a_2 + 100$
$\text{TOT} \leftarrow \text{TOT} + 100$

Transaction: collection of actions that preserve consistency
(= process)

SQL: each query or modification statement
Embedded SQL: Sequence of DB operations up to COMMIT or ROLLBACK ("abort")

Big assumption:
If transaction $T$
- starts with consistent state +
- executes in isolation
⇒ $T$ leaves consistent state

Correctness (informally)
- If we stop running transactions, DB left consistent
- Each transaction sees a consistent DB

How can constraints be violated?
- Transaction bug
- DBMS bug
- Hardware failure
e.g., disk crash alters balance of account
- Simultaneous transactions accessing
  shared data
e.g.: $T_1$: give 10% raise to programmers
  $T_2$: change programmers ⇒ systems analysts

How can we prevent/fix violations?
- This lecture: due to system failures only [Chapter 8]
- Chapter 9: due to concurrency only
- Chapter 10: due to failures and concurrency

Will not consider:
- How to write correct transactions
- How to write correct DBMS
- Constraint checking & repair
  That is, solutions studied here do not need to know constraints
### Failure Model

<table>
<thead>
<tr>
<th>Events</th>
<th>Desired</th>
<th>Undesired</th>
<th>Expected</th>
<th>Unexpected</th>
</tr>
</thead>
</table>

### Context of failure model

- **CPU** - processor
- **memory** - volatile, disappears when power off
- **disk** - nonvolatile

### Desired events:
- see product manuals....

### Undesired expected events:
- System failures:
  - SW errors, power loss
  - memory and transaction state lost

### Undesired Unexpected:
- Everything else!

### Examples:
- Disk data is lost
- Memory lost without CPU halt
- Aliens attack and destroy the building....

### Is this model reasonable?

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

- Replicate disk storage (stable store)
- Memory parity
- CPU checks

### Interacting Address Spaces:

**Storage hierarchy**

- **Input (x):** block with x from disk to buffer
- **Output (x):** block with x from buffer to disk
- **Read (x,t):** transaction variable t := X; (Performs Input(x) if needed)
- **Write (x,t):** X (in buffer) := t

(can protect against this by archive backups)
Note on "database elements" X:

- Anything that can have value and be accessed or modified by transactions:
  - a relation (or extent of an object class)
  - a disk block
  - a record
- Easiest to use blocks as database elements

Key problem

Unfinished transaction

Example

Constraint: A = B \(^*\)

\[ T_1: \ A \leftarrow A \times 2 \]
\[ B \leftarrow B \times 2 \]

\(^*\) simplification; a more realistic example: the sum of loan balances = the total debt of a bank

One "complication"

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

Undo logging

Immediate modification

\[ T_1: \] Read (A,t); t \leftarrow t \times 2
Write (A,t);
Read (B,t); t \leftarrow t \times 2
Write (B,t);
Output (A);
Output (B);

\[ A: 8 \]
\[ B: 8 \]

\[ A: 16 \]
\[ B: 16 \]

memory

Disk

Log

\[ A: 8 \]
\[ B: 8 \]

DB

BAD STATE

\# 1
One “complication”

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

Memory
A: 16
B: 8

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

DB
BAD STATE
# 2

Undo logging rules

1. Generate an undo log record for every update action (with the old value)
2. Before modifying element X on disk, any log records for X must be flushed to disk (write-ahead logging)
3. All WRITEs of transaction T must be OUTPUT to disk before <T, commit> is flushed to log

Recovery rules:

- 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

Recovery with Undo logging

T1: A ← A+1; B ← B+1; \( A=B \)
T2: A ← A+1; B ← B+1;

DB on disk
A: 10
B: 11

Log
<T1, start>
<T1, A, 10>
<T2, start>
<T2, A, 11>

CRASH =>
A should be restored to 10, not 11

Undo logging Recovery:

(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 \(\rightarrow\) earliest) do:
  - If Ti \(\in\) S then
    - Write (X, v)
    - Output (X)
(3) For each Ti \(\in\) S do
  - Write <Ti, abort> to log

What if system fails during recovery?

- No problem!
- Undo idempotent
  - Can repeat (unfinished) undo, result remains the same
To discuss:
• Redo logging
• Undo/redo logging, why both?
• Checkpointing
• Media failures

Redo logging
(Deferred modification)
T1:
Read(A, t); t \times 2; write (A, t);
Read(B, t); t \times 2; write (B, t);
Output(A); Output(B)

Redo logging rules
(1) For every disk update, generate a
redo log record (with new value)
(2) Before modifying DB element X on
disk, all log records for the transaction
that (including COMMIT) must be
flushed to disk

Recovery rules:
Redo logging
(1) For every Ti with <Ti, commit> in log:
- For all <Ti, X, v> in log:
  Write(X, v)
  Output(X)

Recovery with Redo Logging:
(more precisely)
(1) Let S = set of transactions with
<Ti, commit> in log
(2) For each <Ti, X, v> in log, in
forward order (earliest \rightarrow latest) do:
  - if Ti \in S then
    Write(X, v)
    Output(X)

Recovery is very, very SLOW!

Redo log:
First Report
Record T1 wrote A, B
Committed a year ago
Last Record
(1 year ago) \rightarrow STILL, Need to redo after crash!!
**Solution:** Checkpointing  
(for redo-logging, simple version)

**Periodically:**  
1. Stop accepting new transactions  
2. Wait all transactions to finish (commit/abort)  
3. Flush all log records to disk (log)  
4. Flush all buffers to disk (DB)  
5. Write & flush a <CKPT> record on disk (log)  
6. Resume transaction processing

**Example: what to do at recovery?**

**Redo log (on disk):**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Crash</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>commit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>B</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>commit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>C</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result of transactions completed prior to the checkpoint are stored permanently on disk

- > Redo write(B, 17)  
  output(B)

**Drawbacks:**

- **Undo logging:**  
  - System forced to update disk at the end of transactions (may increase disk I/O)  
  - cannot redo recent updates to refresh a backup copy of the DB

- **Redo logging:**  
  - need to keep all modified blocks in memory until commit (may lead to shortage of buffer pool)

**Solution:** Undo/redo logging!

**Update record**

\[ \langle T_i, X, \text{Old-X-value}, \text{New-X-value} \rangle \]

Provides more flexibility to order actions

**Rules for undo/redo logging**

1. Log record  
   \[ \langle T_i, X, \text{old}, \text{new} \rangle \]  
   flushed to disk before writing X to disk

2. Flush the log at commit  
   - Element X can be written to disk either before or after the commit of Ti

**Undo/Redo Recovery Policy**

- To recover from a crash,  
  - **redo** updates by any *committed* transactions (in forward-order, earliest first)  
  - **undo** updates by any *incomplete* transactions (in backward-order, latest first)
Problem with Simple Checkpointing

- Simple checkpointing stops the system from accepting new transactions → system effectively shuts down
- Solution: "nonquiescent" checkpointing
  - accepts new transactions during a checkpoint

Nonquiescent Checkpointing

- Slightly different for various logging policies
- Rules for undo/redo logging:
  - Write log record (and flush log)
  - Flush to disk all dirty buffers which contain changed database elements (Corresponding log records first)
  - Write & flush log record

Examples: what to do at recovery time?

- Undo T1 (restore Y to b and X to a)

Recovery process:

- Backwards pass (end of log to 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 to end of log)
  - redo actions of transactions in set S
Media failure (loss of non-volatile storage)

Solution: Make copies of data!

Solution a: Use mirrored or redundant disks

- Mirroring: 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
  - if ok, done
  - else try another one

Assumes bad data can be detected

Solution b: Archiving

- If active database is lost,
  - restore active database from backup (archive)
  - bring up-to-date using redo entries in log

When can we discard the log?

Summary

- Consistency of data
- Recovery from system failures
  - Logging (undo/redos/undo-redos), checkpoints
  - Recovery: using log to reconstruct the DB
- Recovery form media failures
  - reducing risk through redundancy
  - backups / archiving
- Another source of problems:
  - Concurrent transactions that access shared DB elements