Recovery from failures

By Marina Barsky
Winter 2016, University of Toronto
Integrity or correctness of data

We would like data to be “accurate” or “correct” at all times

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>52</td>
</tr>
<tr>
<td>Green</td>
<td>3421</td>
</tr>
<tr>
<td>Gray</td>
<td>1</td>
</tr>
</tbody>
</table>
Integrity or consistency constraints

• Predicates that data must satisfy

For example:
- x is key of relation R
- \( x \rightarrow y \) (func. dependency) holds in R
- Domain(x) = \{Red, Blue, Green\}
- no employee should make more than twice the average salary
Integrity constraints may not capture "full correctness"

*Implicit (business) constraints:*
- When salary is updated,  
  new salary > old salary
- When account record is deleted,  
  balance = 0
Definition:

- **Consistent state**: satisfies all constraints
- **Consistent DB**: DB in consistent state
Observation:
DB cannot be consistent *always*

Example: \( a_1 + a_2 + \ldots + a_n = TOT \) (constraint)

Deposit $100 in \( a_2 \):
\[
\begin{align*}
a_2 & \leftarrow a_2 + 100 \\
TOT & \leftarrow TOT + 100
\end{align*}
\]
**Transaction**: collection of actions that bring DB from one consistent state to another

If T starts with consistent state + T executes in isolation

⇒ T leaves in a consistent state

We learned how to ensure that concurrent (interleaving) actions appear as if each transaction runs in isolation
When we may end up with an inconsistent DB?

• Erroneous data entry
• Transaction bug (application programmer error)
• DBMS bug (DBMS programmer error)
• Other program bug (overrides memory page)
• System and media failures
  • power loss
  • memory failure
  • processor stop
  • disk crash
  • catastrophic failure: earthquake, flood, end of world
Coping with system failures

- Logging (undo, redo, ARIES)
- Recovery using log
- Checkpointing
- Redundancy:
  - Replicate disk storage (RAID)
  - Memory parity
  - Archiving
Primitive operations of transactions

There are 3 important address spaces:

1. The disk blocks
2. The main memory (buffer) pages
3. The local variables of a Transaction
Operations:

• **Between buffer and disk:**
  • Input (x): block containing x → memory buffer
  • Output (x): block containing x → disk

• **Between transaction and buffer pages:**
  • Read (x,t): do input(x) if necessary
t ← value of x in page
  • Write (x,t): do input(x) if necessary
    value of x in page ← t
Example: effect of transaction on state of memory and disk

A=8
B=6
Constraint: A=B

<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2. t := t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3. WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4. READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>5. t := t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>6. WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7. OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>8. OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

T1: A ← A × 2
    B ← B × 2
Example: effect of transaction on state of memory and disk

A=8
B=6
Constraint: A=B

<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. READ (A, t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2. t := t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3. WRITE (A, t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4. READ (B, t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>5. t := t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>6. WRITE (B, t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7. OUTPUT (A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>8. OUTPUT (B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

T1: A ← A × 2
    B ← B × 2
T1: 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);
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 16
B: 8 16

memory

A: 8
B: 8
disk
T1: 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);

---

failure!

A: 8 16
B: 8 16

memory

Inconsistent DB!

A: 8 16
B: 8
disk
How to prevent an inconsistent state?

• We cannot prevent an inconsistent state, but we can arrange for the problem to be repaired

• Running the transaction again may not fix the problem

• Need atomicity: execute
  • all actions of a transaction
  • or none at all

Solution 1:
undo logging (immediate modification on disk)
The **transaction manager** will send messages about actions of transactions to the **log manager**, to the **buffer manager** (when to copy the buffer back to disk), and to the **query processor**.

The **log manager** maintains the log. It must deal with the buffer manager, since space for the log initially appears in main-memory buffers, and at certain times these buffers must be copied to disk.
Log records

- A log is a file opened for append only
- It consists of log records, each telling something about what some transaction has done.

Log records:

<T, START>: This record indicates that transaction T has begun.

<T, COMMIT>: Transaction T has completed successfully and will make no more changes to database elements.

<T, ABORT>: Transaction T could not complete successfully.

<T, X, v>: Transaction T has changed database element X, and its former value was v.
Undo logging rules

(1) For every action generate update 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 written to log, all writes of transaction must be reflected on disk (forced to disk)

This is called force rule
Undo log - must write to disk in the following order:

1. The **log records** indicating changed database elements.

2. The changed **database elements** themselves.

3. The **COMMIT log record**.

![Diagram showing the order of log records](attachment:image.png)
Example: Undo logging (Immediate 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

disk

log
Example: Undo logging (Immediate modification)

T1: 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);

\[
\begin{align*}
\text{memory} & : A:8 \quad \text{16} \quad B:8 \quad \text{16} \\
\text{disk} & : A:8 \quad B:8 \\
\text{log} & : \langle T1, \text{start} \rangle \\
& \quad \langle T1, A, 8 \rangle \\
& \quad \langle T1, B, 8 \rangle
\end{align*}
\]
Example: Undo logging (Immediate 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
disk
A:8  B:8
log

<T1, start>
<T1, A, 8>
<T1, B, 8>
**Example: Undo logging** *(Immediate modification)*

**T1:**
- 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 \((B)\);

\(A = B\)

\[-\]

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

\(\text{memory} \quad \text{disk} \quad \text{log}\)
Example: Undo logging (Immediate 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
A:8
B:8

memory

disk

log

<T1, start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>
Flushed log to disk: explicitly

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

<table>
<thead>
<tr>
<th>MEMORY</th>
<th>DB</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 8 16</td>
<td>A: 8</td>
<td></td>
</tr>
<tr>
<td>B: 8 16</td>
<td>B: 8</td>
<td></td>
</tr>
</tbody>
</table>

Log:
- \(<T_1, \text{start}>\)
- \(<T_1, A, 8>\)
- \(<T_1, B, 8>\)
Flushing log to disk: explicitly

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

MEMORY

A: 8 16
B: 8 16
Log:
<T₁, start>
<T₁, A, 8>
<T₁, B, 8>

DB

A: 8 16
B: 8

LOG

BAD STATE: cannot recover!

Database changed before log records reached disk
Order of steps and disk writes in case of UNDO log

<table>
<thead>
<tr>
<th>Step</th>
<th>Activity</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T,START&gt;</td>
</tr>
<tr>
<td>2)</td>
<td>READ (A, t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>3)</td>
<td>t := t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td>WRITE (A, t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>5)</td>
<td>READ (B, t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6)</td>
<td>t := t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7)</td>
<td>WRITE (B, t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>8)</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9)</td>
<td>OUTPUT (A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10)</td>
<td>OUTPUT (B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12)</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T,COMMIT&gt;</td>
</tr>
</tbody>
</table>
Recovery using UNDO log

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

Because multiple uncommitted transactions could potentially modify the same element several times, the undo operations are in reverse order (latest → earliest)
What if failure during recovery?

No problem!  Undo idempotent
Example: Recovery using Undo log

<table>
<thead>
<tr>
<th>Step</th>
<th>Activity</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T, START&gt;</td>
</tr>
<tr>
<td>2</td>
<td>READ (A, t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>t := t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>WRITE (A, t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T, A, 8&gt;</td>
</tr>
<tr>
<td>5</td>
<td>READ (B, t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>t := t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>WRITE (B, t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T, B, 8&gt;</td>
</tr>
<tr>
<td>8</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>OUTPUT (A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>OUTPUT (B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T, COMMIT&gt;</td>
</tr>
<tr>
<td>12</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The crash occurs after step (12). Then the <COMM IT T> record reached disk before the crash. When we recover, we do not undo the results of T, and all log records concerning T are ignored by the recovery manager.
Example: Recovery using Undo log

<table>
<thead>
<tr>
<th>Step Activity</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2) READ (A, t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T, START&gt;</td>
</tr>
<tr>
<td>3) t := t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4) WRITE (A, t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>5) READ (B, t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6) t := t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7) WRITE (B, t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>8) FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9) OUTPUT (A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10) OUTPUT (B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T, COMMIT&gt;</td>
</tr>
<tr>
<td>12) FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The crash occurs between steps (11) and (12). If <COMMIT T> record reached disk see previous case, if not, see next case.
The crash occurs between steps (10) and (11). Now, the COMMIT record surely was not written, so \( T \) is incomplete and is undone.
**Example: Recovery using Undo log**

<table>
<thead>
<tr>
<th>Step</th>
<th>Activity</th>
<th>$t$</th>
<th>$M-A$</th>
<th>$M-B$</th>
<th>$D-A$</th>
<th>$D-B$</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$&lt;T,\text{START}&gt;$</td>
</tr>
<tr>
<td>2)</td>
<td>READ($A,t$)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>$&lt;T,A,8&gt;$</td>
</tr>
<tr>
<td>3)</td>
<td>$t := t \times 2$</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td>WRITE($A,t$)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5)</td>
<td>READ($B,t$)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6)</td>
<td>$t := t \times 2$</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7)</td>
<td>WRITE($B,t$)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>$&lt;T,B,8&gt;$</td>
</tr>
<tr>
<td>8)</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9)</td>
<td>OUTPUT($A$)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10)</td>
<td>OUTPUT($B$)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$&lt;T,\text{COMMIT}&gt;$</td>
</tr>
<tr>
<td>12)</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The crash occurs between steps (8) and (10). Again, $T$ is undone. In this case the change to $A$ and/or $B$ may not have reached disk. Nevertheless, the proper value, 8, is restored for each of these database elements.
The crash occurs prior to step (8). Now, it is not certain whether any of the log records concerning T have reached disk. If the change to A and/or B reached disk, then the corresponding log record reached disk. Therefore if there were changes to A and/or B made on disk by T, then the corresponding log record will cause the recovery manager to undo those changes.
Log needs to be truncated

- Log can become larger than DB itself
- It takes too much time to check all the log records when recovery is needed
- We want to truncate some old log records, which are no longer needed
- Can we delete everything prior to `<T, COMMIT>`?

No, because the actions of some uncommitted transactions are interleaving

Solution: checkpointing
Quiescent checkpointing

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

Every transaction executed before checkpoint has finished and the log can be truncated

Problem: while waiting for all active transactions to complete, DB appears stalled to its users

Solution: non-quiescent checkpointing
Non-quiescent checkpointing

1. Write log record <START CKPT(T1, ...Tk)>
   T1 ... Tk are active transactions, and flush log.

2. Wait until all T1 ... Tk commit or abort, but don’t prohibit other transactions from starting.

3. When all T1 ... Tk have completed, write a log record <END CKPT> and flush the log.
Recovery using undo – start checkpoint

Transactions with commit in log

Active (started not committed) transactions T1...Tk

All committed transactions are already on disk

<START CKPT T1...Tk>

memory
disk
log

WAL log
COMMIT
Updated DB blocks

UNDO logging
Recovery using undo – during checkpoint

- Transactions with commit in log
  - Active (started not committed) transactions T1...Tk
  - New transactions started during checkpoint

- All committed transactions are already on disk
  - Some of T1...Tk commit
  - Some of T1..Tk are active, partly can be written to disk

- Updated DB blocks

WAL log

<START CKPT T1...Tk>
Recovery using undo – during checkpoint

- All committed transactions are already on disk
- Some of T1...Tk commit
- Some of T1..Tk are active
- Some of new transactions commit
- Some of new are active

Transactions with commit in log

Active (started not committed) transactions T1...Tk

New transactions started during checkpoint

memory

disk

log

WAL log

Updated DB blocks

UNDO logging

<START CKPT T1...Tk>
Recovery using undo – failure during checkpoint

Transactions with commit in log
- Active (started not committed) transactions T1...Tk
- New transactions started during checkpoint

All committed transactions are already on disk
- Some of T1...Tk commit
- Some of T1..Tk are active
- Some of new transactions commit
- Some of new are active

WAL log
- COMMIT
- UNDO logging

Updated DB blocks

<START CKPT T1...Tk>

Undo these
- Scan backwards pass START CKPT only for uncommitted among T1...Tk
Recovery using undo – end checkpoint

Transactions with commit in log

Active (started not committed) transactions T1...Tk

New transactions started during checkpoint

---

All committed transactions are already on disk

ALL T1...Tk commit

Some of new transactions commit

Some of new are active

---

<START CKPT T1...Tk>

<END CKPT>

---

WAL log

COMMIT

UNDO logging

Updated DB blocks

---

memory
disk
log
Recovery using undo – failure after end checkpoint

All committed transactions are already on disk

Transactions with commit in log
Active (started not committed) transactions T1...Tk

New transactions started during checkpoint

Undo only new transactions commit
Some of new are active

<START CKPT T1...Tk>

Updated DB blocks
WAL log
COMMIT

<END CKPT>

Log before START CKPT can be deleted

memory

disk

log
Recovery using UNDO log with checkpointing – in words

Scanning log backwards:

• If we first meet an `<END CKPT>` record, then we know that all incomplete transactions began after the matching `<START CKPT (T1, ..., Tk)>` record.

We may thus scan backwards as far as this `<START CKPT>`, and then stop; previous log is useless and may be discarded after the recovery.

• If we first meet a record `<START CKPT (T1, ..., Tk)>`, then the crash occurred during the checkpoint. We need scan no further back than the start of the earliest of these incomplete transactions.

• General rule: once `<END CKPT>` is written, we can discard the log prior to the preceding `<START CKPT>` record.
Problems with UNDO logging

• The buffer pages forced to disk before writing `<COMMIT T>`, at the time that could be not the best from the disk performance perspective

• Too many disk I/Os

• How can we save disk I/Os allowing changed data reside in memory buffers for a while?

Solution 2: Redo logging
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
Redo log - must write to disk in the following order:

1. The log records indicating changed database elements.
2. The COMMIT log record.
3. The changed database elements themselves.

The changes remain in buffer until COMMIT log record reaches disk. That means that we cannot free dirty pages, until transaction is complete, we cannot steal them – this is called no steal rule.
Example: 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)
Example: 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

<T1, start>
<T1, A, 16>
<T1, B, 16>
<T1, commit>

LOG
Example: Redo logging (deferred modification)

T1:  
Read(A,t); t × 2; write (A,t);
Read(B,t); t × 2; write (B,t);
Output(A); Output(B)

memory

memory

DB

LOG

A: 8 16
B: 8 16

output

A: \cancel{8} 16
B: \cancel{8} 16

<T1, start>
<T1, A, 16>
<T1, B, 16>
<T1, commit>
Order of steps and disk writes in case of REDO log

<table>
<thead>
<tr>
<th>Step</th>
<th>Activity</th>
<th>t</th>
<th>M-A</th>
<th>M-B</th>
<th>D-A</th>
<th>D-B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T,START&gt;</td>
</tr>
<tr>
<td>2)</td>
<td>READ (A, t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>3)</td>
<td>t := t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td>WRITE (A, t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>5)</td>
<td>READ (B, t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6)</td>
<td>t := t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7)</td>
<td>WRITE (B, t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>8)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;T,COMMIT&gt;</td>
</tr>
<tr>
<td>9)</td>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10)</td>
<td>OUTPUT (A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>11)</td>
<td>OUTPUT (B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

After logging commit
Recovery using REDO log

For each Ti with <Ti, commit> in log:
   For all <Ti, X, v> in log:
      Write(X, v)
      Output(X)
For each Ti without commit, write <Ti, abort>

Because we need to replay committed transactions in the order they were executed, the redo operations are in forward order (earliest → latest)
REDO log with non-quiescent checkpointing

1. Write log record <START CKPT(T1, ...Tk)> where T1 ... Tk are active (uncommitted) transactions, and flush log.

2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the <START CKPT> record was written to the log. (dirty buffers only of committed transactions)

3. When writing of all pages from committed transactions is complete, write a log record <END CKPT> and flush the log.
Recovery using redo – start checkpoint

Transactions with commit in log
- Active (started not committed) transactions T1...Tk

Some of committed transactions are on disk
- Some of committed transactions are partly on disk
- None of T1...Tk are on disk

<START CKPT T1...Tk>

WAL log
COMMIT

Updated DB blocks

REDO logging
Recovery using redo – during checkpoint

- **Transactions with commit in log**
  - Active (started not committed) transactions T1...Tk
  - New transactions started during checkpoint

- **Some of committed transactions are on disk**
- **Some of committed transactions are partly on disk**
- **Some of T1...Tk commit and are fully or partly on disk**

**Memory**

**Disk**

**Log**
Recovery using redo – during checkpoint

Transactions with commit in log
Active (started not committed) transactions T1...Tk
New transactions started during checkpoint

Some of committed transactions are on disk
Some of committed transactions are partly on disk
Some of T1...Tk commit and are fully or partly on disk
Some of new transactions commit and are fully or partly on disk

<START CKPT T1...Tk>

WAL log
Updated DB blocks
REDO logging
Recovery using redo – failure during checkpoint

- Transactions with commit in log
- Active (started not committed) transactions T1...Tk
- New transactions started during checkpoint

Transactions with commit in log:
- Some of committed transactions are on disk
- Some of committed transactions are partly on disk
- Some of T1...Tk commit and are fully or partly on disk
- Some of new transactions commit and are fully or partly on disk

New transactions started during checkpoint:
- We need to redo all these – no way of knowing which committed are written to disk – scan to the previous END CKPT
Recovery using redo – end checkpoint

- Transactions with commit in log
- Active (started not committed) transactions T1...Tk
- New transactions started during checkpoint

All dirty buffers with transactions committed before checkpoint are written to disk

Some of T1...Tk commit and are fully or partly on disk

Some of new transactions commit and are fully or partly on disk

<START CKPT T1...Tk>

WAL log

COMMIT

Updated DB blocks

REDO logging

<END CKPT>
Recovery using redo – failure after end checkpoint

Transactions with commit in log
Active (started not committed) transactions T1...Tk
New transactions started during checkpoint

All dirty buffers with transactions committed before checkpoint are written to disk
Some of T1...Tk commit and are fully or partly on disk
Some of new transactions commit and are fully or partly on disk

We need to redo only committed among new and T1...Tk. So scan no further than first START T1...Tk

WAL log
Updated DB blocks

COMMIT
REDO logging

memory
disk
log
Recovery with a checkpointed REDO – in words

Scanning log backwards:

• If we first meet an \(<\text{END CKPT} (T_1, T_2, \ldots, T_k)\>\), we need to scan no further back than \(<\text{start T}>\) log record of the earliest of \(T_1, T_2, \ldots, T_k\). We know that all other committed transactions were flushed to disk before END CKPT was written. We limit our attention to \(T_1, T_2, \ldots, T_k\) and any other transactions that started after the matching \(<\text{START CKPT}\>

• If we first meet a record \(<\text{START CKPT} (T_1, \ldots, T_k)\>\), then the crash occurred during the checkpoint. We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk. Thus, we must search back to the previous \(<\text{END CKPT}\>\) record, find its matching \(<\text{START CKPT} (T_1, \ldots, T_m)\>\), and replay all the transactions that started after it or are among \((T_1, \ldots, T_m)\)
Key drawbacks

- *Undo logging*: need frequent disk writes
- *Redo logging*: need to keep all modified blocks in memory until commit

Solution: undo/redo logging (ARIES) – increased flexibility at the expense of larger log
Undo/redo logging

Update ⇒ <Ti, X, Old X val, New X val>

• Page with X can be flushed before or after <COMMIT T> is written
• Log record has to be flushed before corresponding updated page (WAL)
• Flush log after <COMMIT T> is written (solves problem of delayed commitment)
Undo/redo logging rules

**UR1** Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record \(<T, X, v, w>\) appear on disk.

**UR2** A \(<\text{COMMIT T}>\) record must be flushed to disk as soon as it appears in the log.
Undo/redo recovery policy

1. Redo all the committed transactions in the order earliest-first, and
2. Undo all the uncommited transactions in the order latest-first.
Checkpointing for undo/redo logging

1. Write log record `<START CKPT(T1, ...Tk)>`
   
   T1 ... Tk are active (uncommitted) transactions, and flush log.

2. Write to disk all dirty buffers – not just committed

3. When all dirty buffers are written, write a log record `<END CKPT>` and flush the log.

The flexibility of undo/redo logging allows to write to disk changes that has not yet been committed – we can undo them in case of failure.
Recovery using undo/redo – start checkpoint

Transactions with commit in log
Active (started not committed) transactions T1...Tk

Committed transactions are fully or partly on disk
Some of T1...Tk are fully or partly on disk

memory

disk

log

WAL log
COMMIT

UNDO/REDO logging

Updated DB blocks

<START CKPT T1...Tk>
Recovery using undo/redo — start checkpoint

Transactions with commit in log
Active (started not committed) transactions T1…Tk
Some of T1…Tk commit
New transactions start

Committed transactions are fully or partly on disk
Some of T1…Tk are fully or partly on disk
Some of new transactions are fully or partly on disk

<START CKPT T1…Tk>

WAL log
COMMIT
Updated DB blocks

UNDO/REDO logging

memory

disk

log
Recovery using undo/redo – failure during checkpoint

- Transactions with commit in log
- Active (started not committed) transactions T1...Tk
- Some of T1...Tk commit
- New transactions start

Committed transactions are fully or partly on disk

Some of T1...Tk are fully or partly on disk

Some of new transactions are fully or partly on disk

We need to redo all committed and undo all uncommitted, including everything. Scan to the previous END CKPT
Recovery using undo/redo – end checkpoint

Transactions with commit in log

Active (started not committed) transactions T1...Tk

Some of T1...Tk commit

New transactions start

Committed transactions are fully on disk

Some of T1...Tk are fully or partly on disk

Some of new transactions are fully or partly on disk

<START CKPT T1...Tk>

<END CKPT>

WAL log

Updated DB blocks

UNDO/REDO logging

memory

disk

log
Recovery using undo/redo – end checkpoint

- Transactions with commit in log
- Active (started not committed) transactions T1...Tk
- Some of T1...Tk commit
- New transactions start

Committed transactions are fully on disk

Some of T1...Tk are fully or partly on disk

Some of new transactions are fully or partly on disk

WAL log
Updated DB blocks

<START CKPT T1...Tk>

We need to redo and undo only new and T1...Tk. Scan no further back than START of the earliest of T1...Tk

<END CKPT>
Example: Recovery using Undo/Redo log

Log on disk

<START T1>
<T1,A,4,5>
<START T2>
<COMMIT T1>
<T2,B,9,10>
<START CKPT (T2)>
<T2,C,14,15>
<START T3>
<T3,D,19,20>
<END CKPT>
<COMMIT T3>
<COMMIT T3>
<COMMIT T2>

- When checkpoint begins, the only active transaction is T2
- In any case, we flush to disk pages with A and B, if they are still in buffer
- During the checkpoint, we flush A of T1, B of T2 (all dirty pages)
- New transaction T3 starts during the checkpoint
Example: Recovery using Undo/Redo log

Log on disk

<START T1>
<T1,A,4,5>
<START T2>
<COMMIT T1>
<T2,B,9,10>
<START CKPT (T2)>
<T2,C,14,15>
<START T3>
<T3,D,19,20>
<END CKPT>
<COMMIT T3>
<COMMIT T2>

• If the crash at the end, T2 and T3 are committed and are redone.
• Because we find <END CKPT>, we know that all transactions committed before <START CKPT> are flushed to disk.
• When we redo T2, we only consider log records after <START CKPT>, because we know that during checkpoint all dirty pages (including B) were flushed to disk.
Example: Recovery using Undo/Redo log

Log on disk

<START T1>
<T1,A,4,5>
<START T2>
<COMMIT T1>
<T2,B,9,10>
<START CKPT (T2)>
<T2,C,14,15>
<START T3>
<T3,D,19,20>
<END CKPT>
<COMMIT T3>
<COMMIT T2>

- Because we find <END CKPT>, we know that all transactions committed before <START CKPT> are flushed to disk
- There are no other committed transactions, so we need to undo T3 (which started during the checkpoint)
- We also need to undo all actions of T2, scanning log past <START CKPT> until <START T2>
Example: Recovery using Undo/Redo log

Log on disk

<START T1>
<T1,A,4,5>
<START T2>
<COMMIT T1>
<T2,B,9,10>
<START CKPT (T2)>
<T2,C,14,15>
<START T3>
<T3,D,19,20>
<END CKPT>
<COMMIT T3>
<COMMIT T2>

• Because we find <END CKPT>, we know that all transactions committed before <START CKPT> are flushed to disk

• In this case, we redo all actions of T3, and undo all actions of T2
Example: Recovery using Undo/Redo log

Log on disk

<START T1>
<T1,A,4,5>
<START T2>
<COMMIT T1>
<T2,B,9,10>
<START CKPT (T2)>
<T2,C,14,15>
<START T3>
<T3,D,19,20>
<END CKPT>

• Because we do not find <END CKPT> in the log, we do not know whether transactions committed before <START CKPT> are flushed to disk or not

• We need to redo all committed transactions (in this case only T1)

• We need to undo all uncommitted transactions (T2 and T3)

• We will scan the log until the previous checkpointing block – in this example until the beginning of the file
Media failure
(loss of non-volatile storage)

A: 16

Solution: Make copies of data!
Triple modular redundancy

- Keep 3 copies on separate disks
- Output($X$) --> three outputs
- Input($X$) --> three inputs + vote
If active database is lost,
   – restore active database from backup
   – bring up-to-date using redo entries in log
Non-quiescent archiving

• 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]
```

• To restore – we need the dump and the log created during the backup
Summary

• To preserve DB consistency: need mechanisms to get out of an inconsistent state created due to failure
• Two main recovery techniques: logging and redundant copies
• The most flexible logging protocol: undo/redo
• Checkpoints prevent log from indefinite growth
Mechanisms that guarantee ACID transactions

- **Atomicity**: recovery with undo/redo logging
- **Consistency**: serializable schedules, logging for the event of crash
- **Isolation**: serializable schedules, locking
- **Durability**: write-ahead logging, redundant copies