CompSci 516
Database Systems

Lecture 15
Transactions – Recovery

Instructor: Sudeepa Roy
Announcements (Thurs 2/24)

• HW2-part 1 due next week 3/3 (Thursday) 12 noon
  – Extended deadline by two days
  – One group submission per pair is needed
  – Part 2 (on cloud) will be released later in the semester if we have time due to change in AWS setups
  – If you are still looking for someone to work with, send me an email NOW by 12 noon today!
  – Shweta had a tutorial yesterday, watch the recording

• Midterm Project report due next week 3/4 (Friday) 12 noon
  – Extended by ~3 days
  – Keep working on your project
Reading Material

• [GUW]
  – 17.4: UNDO/REDO
  – Lecture slides will be sufficient for exams

Acknowledgement:
A few of the following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Today

Recovery

• STEAL/ NO STEAL
• FORCE/NO FORCE
• UNDO/REDO log
• Checkpointing and Recovery
Review: The ACID properties

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

- Which property did we cover in CC? : Isolation
- Now : Atomicity and Durability by recovery manager
Motivation: A & D

Commit ≠ Disk Write!
Abort ≠ No Disk Write!
Eventually yes, but not necessarily immediately

- **Atomicity:**
  - Transactions may abort ("Rollback").
- **Durability:**
  - What if DBMS stops running?
  - (power failure/crash/error/fire-flood etc.)

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

• Atomicity
  – by “undo”ing actions of “aborted transactions”

• Durability
  – by making sure that all actions of committed transactions survive crashes and system failure
  – i.e. by “redo”-ing actions of “committed transactions”
Assumptions

• Concurrency control is in effect

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

• Simple schemes to guarantee Atomicity & Durability (next):
  – NO STEAL
  – FORCE
Handling the Buffer Pool

- **Force** every write to disk?

- **Steal** buffer-pool frames from uncommitted transactions?
Handling the Buffer Pool

- **Force every write to disk?**
  - Poor response time
  - But provides durability

- **Not Steal buffer-pool frames from uncommitted transactions?**
  - If not steal, poor throughput, holding on to all dirty blocks requires lots of memory
  - If steal, how can we ensure atomicity?

```
<table>
<thead>
<tr>
<th></th>
<th>No Steal</th>
<th>Steal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Force</td>
<td>Trivial</td>
<td>Desired</td>
</tr>
</tbody>
</table>
```
What if we do “Steal” and “NO Force”

• **STEAL** (why enforcing Atomicity is hard)
  – To steal frame F: Current page in F (say P) is written to disk; some transaction holds lock on P
  – What if the transaction 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.
Basic Idea: Logging

• **Log**: An ordered list of REDO/UNDO actions
  – Log record may contain:
    `<Tr.ID, pageID, offset, length, old data, new data>`

• Record REDO and UNDO information, for every update, in a log

• one change turns into two—bad for performance?
  – Sequential writes to log (put it on a separate disk) – append only
  – Minimal info (diff) written to log, so multiple updates fit in a single log page
  – Log blocks are created and updated in the main memory first, then written to disk
  – Can use dedicated disk(s) to improve performance
Different types of logs

- **UNDO (STEAL + FORCE)**
- **REDO (NO STEAL + NO FORCE)**
- **UNDO/REDO (STEAL + NO FORCE)**

GUW 17.4
(Lecture material will be sufficient for Exams)

- **ARIES**
  - an UNDO/REDO log implementation

Will talk about ARIES if we have time later

We only talk about UNDO/REDO In this lecture
UNDO/REDO logging
UNDO/REDO logging

• Simple representation for illustration
  – (actual implementation has more info)

• \(<T, X, v, w>\)
  – Transaction T changed the value of element X
    – former value v
    – new value w
UNDO/REDO logging rule

When a transaction $T$ starts, log $\langle \text{START } T \rangle$

Before modifying any element $X$ on disk, $\langle T, X, v, w \rangle$ must appear on disk

A transaction $T_i$ is committed when its commit log record $\langle \text{COMMIT } T \rangle$ is written to disk

- can precede or follow any of the changes to the db elements on disk
WAL

• **Write-ahead logging (WAL):** Before $X$ is modified on disk, the log record pertaining to $X$ must be flushed
  – Without WAL, system might crash after $X$ is modified on disk but before its log record is written to disk—no way to undo

• **No force:** A transaction can commit even if its modified memory blocks have not been written to disk (since redo information is logged)

• **Steal:** Modified memory blocks can be flushed to disk anytime (since undo information is logged)
Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)

Memory buffer

Disk

A = 800
B = 400
Undo/redo logging example

\( T_1 \) (balance transfer of $100 from A to B)

Memory buffer

Disk

\[ A = 800 \]
\[ B = 400 \]

Log

\( \langle T_1, \text{start} \rangle \)
Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)

\[
\text{read}(A, a); \ a = a - 100;
\]
Undo/redo logging example

$T_1$ (balance transfer of $100 from A to B)$

read($A, a); a = a - 100;$

Memory buffer

$A = 800$

Disk

$A = 800$

$B = 400$

Log

$\langle T_1, \text{start} \rangle$
Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)

read($A, a$); $a = a - 100$;
write($A, a$);

Memory buffer

A = 800

Disk

A = 800
B = 400

Log

$\langle T_1, \text{start} \rangle$
Undo/redo logging example

$T_1$ (balance transfer of $100 from A to B)

read(A, a); a = a – 100;
write(A, a);

Disk
A = 800
B = 400

Log
$\langle T_1, \text{start} \rangle$
$\langle T_1, A, 800, 700 \rangle$

Memory buffer
A = 800 700
**Undo/redo logging example**

\[ T_1 \] (balance transfer of $100 from \( A \) to \( B \))

```
read(A, a); a = a - 100;
write(A, a);
read(B, b); b = b + 100;
```

<table>
<thead>
<tr>
<th>Memory buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 800 ) 700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = 800 )  ( B = 400 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle T_1, \text{start} \rangle )</td>
</tr>
<tr>
<td>( \langle T_1, A, 800, 700 \rangle )</td>
</tr>
</tbody>
</table>
Undo/redo logging example

\( T_1 \) (balance transfer of $100 from A to B)

read(\( A, a \)); \( a = a - 100 \);
write(\( A, a \));
read(\( B, b \)); \( b = b + 100 \);

\[
\begin{align*}
Memory buffer & \\
A = 800 & \rightarrow 700 \\
B = 400 &
\end{align*}
\]

Disk

\[
\begin{align*}
A = 800 \\
B = 400
\end{align*}
\]

Log

\[
\begin{align*}
\langle T_1, \text{start} \rangle & \\
\langle T_1, A, 800, 700 \rangle
\end{align*}
\]
Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)
read($A, a$); $a = a - 100$
write($A, a$);
read($B, b$); $b = b + 100$
write($B, b$);

Memory buffer
\[
\begin{align*}
A &= 800 \\
B &= 400
\end{align*}
\]

Disk
\[
\begin{align*}
A &= 800 \\
B &= 400
\end{align*}
\]

Log
\[
\langle T_1, \text{start} \rangle \\
\langle T_1, A, 800, 700 \rangle
\]
Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)

read($A$, $a$); $a = a - 100$;
write($A$, $a$);
read($B$, $b$); $b = b + 100$;
write($B$, $b$);

Memory buffer

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$800$</td>
</tr>
<tr>
<td>$B$</td>
<td>$400$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Disk

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$800$</td>
</tr>
<tr>
<td>$B$</td>
<td>$400$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Log

\[
\langle T_1, \text{start} \rangle \\
\langle T_1, A, 800, 700 \rangle \\
\langle T_1, B, 400, 500 \rangle
\]
Undo/redo logging example

\( T_1 \) (balance transfer of $100 from A to B)

read(A, a); \( a = a - 100; \)
write(A, a);
read(B, b); \( b = b + 100; \)
write(B, b);

Steal: can flush before commit
Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)

read($A$, $a$); $a = a - 100$
write($A$, $a$);
read($B$, $b$); $b = b + 100$
write($B$, $b$);
commit;

Memory buffer

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>800</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>500</td>
</tr>
</tbody>
</table>

Disk

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>800</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>700</td>
<td>500</td>
</tr>
</tbody>
</table>

Log

\[
\langle T_1, \text{start} \rangle
\langle T_1, A, 800, 700 \rangle
\langle T_1, B, 400, 500 \rangle
\]

Steal: can flush before commit
Undo/redo logging example

$T_1$ (balance transfer of $100$ from $A$ to $B$)

read($A, a$); $a = a - 100$;
write($A, a$);
read($B, b$); $b = b + 100$;
write($B, b$);
commit;

Steal: can flush before commit
$T_1$ (balance transfer of $100 from $A to $B)

read($A, a$); $a = a - 100$;
write($A, a$);
read($B, b$); $b = b + 100$;
write($B, b$);
commit;

**Memory buffer**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>800</td>
</tr>
<tr>
<td>$B$</td>
<td>400</td>
</tr>
</tbody>
</table>

**Disk**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>700</td>
</tr>
<tr>
<td>$B$</td>
<td>500</td>
</tr>
</tbody>
</table>

**Log**

- $\langle T_1, \text{start} \rangle$
- $\langle T_1, A, 800, 700 \rangle$
- $\langle T_1, B, 400, 500 \rangle$
- $\langle T_1, \text{commit} \rangle$

Steal: can flush before commit

No force: can flush after commit
Undo/redo logging example

$T_{1}$ (balance transfer of $100$ from $A$ to $B$)

read($A$, $a$); $a = a - 100$;
write($A$, $a$);
read($B$, $b$); $b = b + 100$;
write($B$, $b$);
commit;

Steal: can flush before commit

No force: can flush after commit

No restriction (except WAL) on when memory blocks can/should be flushed
Checkpointing

• Where does recovery start? Beginning of very large log file?
  – No – use checkpointing

Naïve approach:
• To checkpoint:
  – Stop accepting new transactions (lame!)
  – Finish all active transactions
  – Take a database dump

• To recover:
  – Start from last checkpoint
Fuzzy checkpointing

• Add to log records `<START CKPT S>` and `<END CKPT>`
  – Transactions normally proceed and new transactions can start during checkpointing (between START CKPT and END CKPT)

• Determine $S$, the set of (ids of) currently active transactions, and log `<START CKPT S>`

• Flush all blocks (dirty at the time of the checkpoint) at your leisure

• Log `<END CKPT START-CKPT_location>`
  – To easily access `<START CKPT>` of an `<END CKPT>` otherwise can read the log backword to find it
An UNDO/REDO log with checkpointing

- **T2 is active, T1 already committed**
  - So `<START CKPT (T2)>`

- **During CKPT,**
  - flush A to disk if it is not already there (dirty buffer)
  - flush B to disk if it is not already there (dirty buffer)
  - Assume that the DBMS keeps track of dirty buffers
Recovery using Log and CKPT:
Three steps at a glance

1. Analysis
   - Runs backward, from end of log, to the <START CKPT> of the last <END CKPT> record found (note this would be encountered “first” when reading backwards)
   - Goal: Reach the relevant <START CKPT> record

2. Repeating history (also completes REDO for committed transactions)
   - Runs forward, from START CKPT, to the end of log
   - Goal: (1) Repeat all updates from START CKPT (whether or not they already went to the disk, whether or not they are from committed transactions), (2) Build set U of uncommitted transaction to be used in UNDO step below

3. UNDO
   - Runs backward, from end of log, to the earliest <START T> of the uncommitted transactions stored in set U (note this may be before or after the <START CKPT> found in analysis step)
   - Goal: UNDO the actions of uncommitted transactions
Recovery: (1) analysis and (2) repeating history/REDO phase

- Need to determine \( U \), the set of active transactions at time of crash
- Scan log backward to find the last <END CKPT> record and follow the pointer to find the corresponding \(<\text{START CKPT S}\>\)
- Initially, let \( U \) be \( S \)
- Scan forward from that start-checkpoint to end of the log
  - For a log record \(< T, \text{start} >\), add \( T \) to \( U \)
  - For a log record \(< T, \text{commit} | \text{abort} >\), remove \( T \) from \( U \)
  - For a log record \(< T, X, \text{old}, \text{new} >\), issue write(\( X, \text{new} \))

\( \triangleright \) Basically repeats history!

REDO is done and committed transactions are all in good shape now!
Still need to do UNDO for aborted/uncommitted transactions
Recovery: (3) UNDO phase

- Scan log backward
  - Undo the effects of transactions in $U$
  - That is, for each log record $\langle T, X, old, new \rangle$ where $T$ is in $U$, issue $\text{write}(X, old)$, and log this operation too (part of the “repeating-history” paradigm)
  - Log $\langle T, \text{abort} \rangle$ when all effects of $T$ have been undone

☞ An optimization
  - Each log record stores a pointer to the previous log record for the same transaction; follow the pointer chain during undo
# Recovery: Example 1

<table>
<thead>
<tr>
<th>Log records</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;START T1&gt;</td>
</tr>
<tr>
<td>&lt;T1, A, 4, 5&gt;</td>
</tr>
<tr>
<td>&lt;START T2&gt;</td>
</tr>
<tr>
<td>&lt;COMMIT T1&gt;</td>
</tr>
<tr>
<td>&lt;T2, B, 9, 10&gt;</td>
</tr>
<tr>
<td>&lt;START CKPT(T2)&gt;</td>
</tr>
<tr>
<td>&lt;T2, C, 14, 15&gt;</td>
</tr>
<tr>
<td>&lt;START T3&gt;</td>
</tr>
<tr>
<td>&lt;T3, D, 19, 20&gt;</td>
</tr>
<tr>
<td>&lt;END CKPT&gt;</td>
</tr>
<tr>
<td>&lt;COMMIT T2&gt;</td>
</tr>
<tr>
<td>&lt;COMMIT T3&gt;</td>
</tr>
</tbody>
</table>

- **T1** has committed and writes are already on disk
- After analysis, **U = S = {T2}**
- **REDO** all actions
- Write **C = 15** (T2)
- **UPDATE** **U** to **{T2, T3}**
- Write **D = 20** (T3)
- **<COMMIT T2>** found: **U = {T3}**
- **<COMMIT T3>** found: **U = {}**
- At the end **U = empty**, do nothing (NO UNDO PHASE)

Assume every log record before crash is on disk
Recovery: Example 2

- T1 has committed and writes are already on disk
- After analysis, U = S = {T2}
- REDO all actions
- Write C = 15 (T2)
- UPDATE U to {T2, T3}
- Write D = 20 (T3)
- <COMMIT T2> found: U = {T3}
  - not necessary to set B to 10 (before END CKPT – already on disk)
- UNDO actions of T3 until its start
- Write D = 19 (T3)

Assume every log record before crash is on disk
## Recovery: Example 3

### Log records

<table>
<thead>
<tr>
<th>Log records</th>
<th>Action</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;START T1&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;T1, A, 4, 5&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;START T2&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;COMMIT T1&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;T2, B, 9, 10&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;START CKPT( T2)&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;T2, C, 14, 15&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;START T3&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;T3, D, 19, 20&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;END CKPT&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;COMMIT T3&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;COMMIT T2&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Analysis

- **T1** has committed and writes are already on disk
- **After analysis, U = S = {T2}**
- **REDO** all actions
- Write **C = 15** (T2)
- **UPDATE** **U** to **{T2, T3}**
- Write **D = 20** (T3)
- **<COMMIT T3> found: U= {T2}**
- **UNDO** actions of **T2** until its start
  - Beyond <START CKPT>!
  - Those changes already went to disk
- Write **C = 14** (T2)
- Write **B = 9** (T2)

### Assume every log record before crash is on disk
A Glimpse at ARIES Data Structures

(Details will be covered if we have time)

Dirty page table

<table>
<thead>
<tr>
<th>pageID</th>
<th>recoveryLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>P500</td>
<td>101</td>
</tr>
<tr>
<td>P600</td>
<td>102</td>
</tr>
<tr>
<td>P505</td>
<td>104</td>
</tr>
</tbody>
</table>

Transaction table

<table>
<thead>
<tr>
<th>transID</th>
<th>lastLSN</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1000</td>
<td>104</td>
<td>Running</td>
</tr>
<tr>
<td>T2000</td>
<td>103</td>
<td>Running</td>
</tr>
</tbody>
</table>

Log

<table>
<thead>
<tr>
<th>prevLSN</th>
<th>tID</th>
<th>pID</th>
<th>Log entry</th>
<th>Type</th>
<th>undoNextLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>T1000</td>
<td>P500</td>
<td>Write A “abc” -&gt; “def”</td>
<td>Update</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>T2000</td>
<td>P600</td>
<td>Write B “hij” -&gt; “klm”</td>
<td>Update</td>
<td>-</td>
</tr>
<tr>
<td>102</td>
<td>T2000</td>
<td>P500</td>
<td>Write D “mnp” -&gt; “qrs”</td>
<td>Update</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>T1000</td>
<td>P505</td>
<td>Write C “tuv” -&gt; “wxy”</td>
<td>Update</td>
<td>-</td>
</tr>
</tbody>
</table>

Buffer Pool

<table>
<thead>
<tr>
<th>pageID</th>
<th>PageLSN</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>P500</td>
<td>103</td>
<td>A = def, D = qrs</td>
</tr>
<tr>
<td>P600</td>
<td>102</td>
<td>B = klm</td>
</tr>
<tr>
<td>P505</td>
<td>104</td>
<td>A = def, D = qrs, E = pq</td>
</tr>
<tr>
<td>P700</td>
<td>-</td>
<td>C = tuv, E = pq</td>
</tr>
</tbody>
</table>

Disk

<table>
<thead>
<tr>
<th>pageID</th>
<th>PageLSN</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>P500</td>
<td>-</td>
<td>A = abc, D = mnp</td>
</tr>
<tr>
<td>P600</td>
<td>-</td>
<td>B = hij</td>
</tr>
<tr>
<td>P505</td>
<td>-</td>
<td>A = def, D = qrs, E = pq</td>
</tr>
<tr>
<td>P700</td>
<td>-</td>
<td>C = tuv, E = pq</td>
</tr>
</tbody>
</table>

Developed at IBM, now used in many DBMS, an actual implementation

Duke CS, Spring 2022