CompSci 516
Database Systems

Lecture 15-16
Transactions – Recovery

Instructor: Sudeepa Roy

Announcements

• Keep working on your project
  – Midterm report due on Wednesday, 11/01

Reading Material

• [GUW]
  – Chapter 17.2.1-17.2.4 (UNDO)
  – Chapter 17.3.1-17.3.4 (REDO – next lecture)
  – 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.

Last Lecture

• Dynamic Database + Phantom Problem
• Multiple-granularity locks
• Alternatives timestamp-based (not lock-based) approaches to CC
  – Optimistic: validation tests
  – Timestamp: RT(O) & WT(O) on each object O
  – Multiversion: multiple versions of each object O with different WT and RT

Today

Recovery
• STEAL/ NO STEAL
• FORCE/NO FORCE
• UNDO log
• REDO log
  – to be continued in the next 1-2 lecture

Transaction Recovery and Logs
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?

Motivation: A & D

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

[Diagram showing desired behavior after system restarts with T1, T2, T3, T4, T5]

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”

DB Architecture for Transactions

- Tr. mgr. issues signals
  - to log mgr. to store log records
  - to buffer mgr. about when it should copy buffer to disk
  - to query processor to execute queries/operations that compromise the transaction
- Log mgr.
  - maintains log, deals with buffer mgr. since first log appears in buffer then it is written to disk
  - Recovery mgr.
    - If there is a crash, it is activated
    - examines the log and repairs data if necessary
  - Note: access to the disk is through the buffer mgr.
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?
  - Poor response time
  - But provides durability
- Steal buffer-pool frames from uncommitted transactions?
  - If not, poor throughput
  - If so, how can we ensure atomicity?

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

- 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 may contain: <Tr.ID, pageID, offset, length, old data, new data>

Different types of logs

- UNDO
- REDO
- UNDO/REDO

- ARIES
  - an UNDO/REDO log implementation

Recall: Log Records

- A file opened for appending only
- Log blocks are created and updated in the main memory first
  - then written to disk

GUW 17.2, 17.3, 17.4 (Lecture material will be sufficient for HWs and Exams)
**UNDO logging**

- Make repair to the database by undoing the effect of transactions that have not finished
  - i.e. uncommitted transactions before a crash or aborted transactions

**Types of UNDO log records**

- `<START T>`: transaction T has begun
- `<COMMIT T>`: T has completed successfully, no more changes will be made
  - Note that seeing `<COMMIT T>` does not automatically ensure that changes have been written to disk, has to be enforced by log manager
- `<ABORT T>`: transaction T could not complete successfully
  - job of the transaction mgr to ensure that changes by T never appear on disk or are cancelled
- `<T, X, v>`: update record for UNDO log
  - T has changed object X, and its former value was v
  - This change normally happened in memory after a WRITE, not after OUTPUT to disk
  - NOTE: we only record the old value, not the new value
  - since UNDO log, while undoing, replace with old value

**UNDO logging rules**

1. (U1) If T modifies X, then log record `<T, X, v>` must be written to disk before the new value of X is written to disk
   - so that the update can be undone using v if there is a crash
2. (U2) If T commits, `<COMMIT T>` must be written to disk after all database elements changed by T are written to disk
   - but as soon thereafter as possible

**Order of write to disk for UNDO log**

- Summarizing two rules:
  1. First, the log records indicating changed DB elements should be written
  2. Second, the changed values of the DB elements should be written
  3. Finally, the COMMIT log record should be written

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>T+1</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
<td>&lt;START&gt;</td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>T</td>
<td>Mem A</td>
<td>Mem B</td>
<td>Disk A</td>
<td>Disk B</td>
<td>Log</td>
</tr>
<tr>
<td>--------</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-----------</td>
</tr>
<tr>
<td>READ(A,i)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>WRITE(A,i)</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Initially A = 8, B = 8

EXAMPLE: UNDO LOG

Duke CS, Fall 2017
CompSci 516: Database Systems

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,i)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>WRITE(A,i)</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Initially A = 8, B = 8

EXAMPLE: UNDO LOG

Duke CS, Fall 2017
CompSci 516: Database Systems

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,i)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>WRITE(A,i)</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Initially A = 8, B = 8

EXAMPLE: UNDO LOG

Duke CS, Fall 2017
CompSci 516: Database Systems

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,i)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>WRITE(A,i)</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Initially A = 8, B = 8

EXAMPLE: UNDO LOG

Duke CS, Fall 2017
CompSci 516: Database Systems
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>( \langle T,A,B \rangle )</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>( \langle T,B,A \rangle )</td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>( \langle T,A,B \rangle )</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>( \langle T,B,A \rangle )</td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>( \langle T,A,B \rangle )</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>( \langle T,B,A \rangle )</td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>( \langle T,A,B \rangle )</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>( \langle T,B,A \rangle )</td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>( \langle T,A,B \rangle )</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>( \langle T,B,A \rangle )</td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>( \langle T,A,B \rangle )</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>( t \times=^2 )</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>( \langle T,B,A \rangle )</td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Recovery with UNDO log

- Scan from the end

- If <COMMIT T> is found in log
  - all changes by T have been written to disk – OK
  - <START T> found but no <COMMIT T>
  - some changes might be written, some not
  - Changes by T on disk have to be UNDONE

- <START T> found but no <COMMIT T>
  - all changes by T have been written to disk – OK
  - <COMMIT T>

- Recall rule 1:
  - “If T modifies X, then log record <T, X, v> must be written to disk before the new value of X is written to disk”
  - v was previous value of X
  - For each such change on disk, there will be a log record on disk as well
  - Reset value of X to v in recovery

Recovery using UNDO logging

- first simple (look at entire log)
- then "checkpointing" (no need to look at entire log)

EXAMPLE: UNDO LOG

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,i)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>i:=2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,i)</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>WRITE(B,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>FLUSH LOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

```
1. If <COMMIT T> has been seen, do nothing
   a) nothing to undo, new value already written
2. Otherwise,
   a) T is incomplete or aborted
   b) Change the value of X to v
3. If <COMMIT T> not found
   a) write <ABORT T>
   b) flush the log
   c) resume normal operation
```

Crash example

1. <T,A,8> found, set A = 8 on disk
2. <ABORT T> found, set B = 8 on disk
3. Change the value of X to v in recovery
4. Resume normal operation

UNDO: order of writing to disk
1. <START T>
2. <T,A,10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>

Crash example 2

1. <T,B,8> found, set B = 8 on disk
2. <COMMIT T> found, set B = 8 on disk
3. Change the value of X to v in recovery
4. Resume normal operation

UNDO: order of writing to disk
1. <START T>
2. <T,A,10> (old value 10)
3. A = 12 (new value 12)
4. <COMMIT T>

- Crash after final flush
- <COMMIT T> already on disks
- All log records by T are ignored by the recovery manager
initially A = 8, B = 8

**EXAMPLE UNDO LOG**

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>START T</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>T,A,B</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>T,A,B</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>T,A,B</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>T,B,B</td>
</tr>
</tbody>
</table>

FLUSH LOG

• Crash before final flush
• <COMMIT T> not on disk
• Go backward, first <T, B, 8> found, set B = 8 on disk
• Then <T, A, 8> is found, set A = 8 on disk

• <START T> found. Nothing else can be found in the log for T. Write <ABORT T>

**EXAMPLE UNDO LOG**

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>START T</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>T,A,B</td>
</tr>
<tr>
<td>READ(B)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>T,A,B</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>T,B,B</td>
</tr>
</tbody>
</table>

FLUSH LOG

• Crash before FIRST flush
• <T, A, 8>, <T, B, 8>, <COMMIT T> not on disk
• By rule U1, A and B not changed on disk - do nothing

Does this UNDO method work if T changes A twice?
A = 16
A = 24?

**Checkpointing for UNDO log**

**Checkpointing Motivation**

• So far, recovery requires every log record to be examined
  • If we have seen <COMMIT T>, no need to examine log records of T
    – all changes already on disk
  • Still, we may not be able to truncate log after one transaction committed
    – log records of other active transactions might be lost
    – always need to scan until the start of the log
  • Explicitly checkpoint the log periodically
    – We can stop scanning the log after certain points

**Checkpointing process**

1. Stop accepting new transactions
2. Wait until all currently active transactions commit or abort, and have written <COMMIT T> or <ABORT T> log record
3. Flush log to disk
4. Write a checkpoint log record <CKPT>, flush the log again
5. Resume accepting transactions
Recovery using Checkpointing for UNDO log

- Do not accept new transaction
- Finish T1, T2
  - they committed
- Then write <CKPT> on log
  - Here T3

Log records
- <START T1>
- <T1, A, 5>
- <T1, B, 10>
- <START T2>
- <T2, C, 15>
- <T2, D, 20>
- <COMMIT T1>
- <COMMIT T2>
- <CKPT>
- <START T3>
- <T3, E, 25>
- <T3, F, 30>

Nonquiescent Checkpointing

- Avoids stalling the system and continues accepting new transactions
  - "quiescent" = in a state of inactivity or dormancy

1. Write <START CKPT(T1,...,Tk)> and flush the log
   - T1,...,Tk are active transactions (have not committed and have not written their changes to disk)
2. Checkpointing continues until all of T1,...,Tk aborts or commits
   - but do not prohibit other new transactions to start
3. When all of T1,...,Tk have completed, write <END CKPT> and flush the log again

Recovery with Nonquiescent Checkpointing for UNDO log

- Scan log from the end (as before)
  - find all incomplete transaction as we go
  - restore values for those transactions (undo)
- If <END CKPT> is met first
  - all incomplete transactions started after <START CKPT,...> can delete log records prior to <START CKPT> once <END CKPT> is written to disk
- If <START CKPT(T1,...,Tk)> is met first
  - crash occurred during the checkpoint
  - incomplete transactions
    - either started after <START CKPT,...>
    - or among T1,...,Tk
  - Scan backward until the earliest <START> of all these transactions

Log records
- <START T1>
- <T1, A, 5>
- <T1, B, 10>
- <START T2>
- <T2, C, 15>
- <START T3>
- <T3, E, 25>
- <COMMIT T2>
- <COMMIT T3>
- <END CKPT>
- <T3, F, 30>

Example: Nonquiescent Checkpointing for UNDO log

- <START CKPT(T1, T2)>
  - since T1, T2 are only active transactions at that point
  - <END CKPT> after both committed
- <START T3> during checkpointing

Log records
- <START T1>
- <T1, A, 5>
- <T1, B, 10>
- <START T2>
- <T2, C, 15>
- <START T3>
- <T3, E, 25>
- <COMMIT T1>
- <COMMIT T2>
- <END CKPT>
- <T3, F, 30>

Recovery with Nonquiescent Checkpointing for UNDO log

- First <T3, F, 30> found
  - restore F to 30 (undo change by T3)
- <END CKPT> found
  - All incomplete transactions started after corresponding <START CKPT>.
- <T3, E, 25> found
  - restore E to 25 (undo change by T3)
- No other records to restore until <START CKPT>...
- Stop there – no further changes

Log records
- <START T1>
- <T1, A, 5>
- <T1, B, 10>
- <START T2>
- <T2, C, 15>
- <START T3>
- <T3, E, 25>
- <COMMIT T1>
- <COMMIT T2>
- <END CKPT>
- <T3, F, 30>
Recovery with Nonquiescent Checkpointing for UNDO log

- Scan backward
  - no <END CKPT> found
  - but <START CKPT(T1, T2)> found
  - also <COMMIT T3> found
- T3 and T2 incomplete transactions
  - T3 already committed
- Scan until the earliest of <START T2> and <START T3>
  - here <START T2>
- Along the way backward
  - restore E to 25 (undo change by T3)
  - restore C to 15 (undo change by T2)
  - in this order
  - then stop at <START T2>

Problems with UNDO logging

- We cannot commit T unless all its changes appear on disk
- Sometimes disk I/Os can be saved if the changes can stay in main memory for a while
  - as long as there is a log to fix things in a crash
- Idea: REDO logging

UNDO vs. REDO

<table>
<thead>
<tr>
<th>UNDO</th>
<th>REDO</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancels (UNDO) the effect of incomplete transactions</td>
<td>ignores incomplete transactions</td>
</tr>
<tr>
<td>ignores committed ones</td>
<td>repeats (REDO) the changes made by committed ones</td>
</tr>
<tr>
<td>requires writing changed elements to disk before the commit log record is written</td>
<td>requires writing changed elements to disk after the commit log record is written</td>
</tr>
<tr>
<td>&lt;T, X, v&gt; v is OLD value of X</td>
<td>&lt;T, X, v&gt; v is NEW value of X</td>
</tr>
</tbody>
</table>

REDO logging

Review: UNDO Log

- STEAL
  - to be able to steal modified pages by a running transaction
  - may have to UNDO for uncommitted transactions
- NO FORCE
  - not to force every write of running transaction to disk
  - may have to REDO for committed transactions

Types of REDO log records

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has completed successfully, no more changes will be made
  - Note that seeing <COMMIT T> does not automatically ensure that changes have been written to disk
  - has to be enforced by log manager
- <ABORT T>
  - transaction T could not complete successfully
  - job of the transaction mgr to ensure that changes by T never appear on disk or are cancelled
- <T, X, v>
  - update record for REDO log
  - T has changed object X, and its NEW value is v
- NOTE: we only record the new value, not the old value (unlike UNDO)
REDO logging rule

(R1) Before modifying any element X on disk, all log records pertaining to this modification, including \(<T, X, v>\) and \(<\text{COMMIT T}>\), must appear on disk

- single “redo rule”
- called the WRITE-AHEAD LOGGING (WAL) rule

Order of write to disk for REDO log

1. First, the log records indicating changed DB elements should be written
2. Second, The COMMIT log record should be written
3. Finally, the changed DB elements should be written

Recovery with REDO log

- Identify committed transactions
  - scan from the end to identify committed transactions
  - make redo changes in the forward direction

- For each log record \(<T, X, v>\)
  - If T is not a committed transaction
    - do nothing
  - If T is committed
    - write the value v of element X

- For each incomplete transaction T
  - write \(<\text{ABORT T}>\)
  - Flush the log
Checkpointing process

1. Write log record <START CKPT(T1, ..., Tk)> where T1,...,Tk are the active (uncommitted) transactions, and flush the log
2. Write to disk all db elements that were written to buffers but not yet to disk by transactions that had already committed before the <START CKPT> record was written to the log
3. Write a log record <END CKPT> to the log and flush the log

Unlike (nonquiescent checkpointing for) UNDO log, we can complete the checkpointing for REDO without waiting for the active transactions to commit or abort, as they are not writing to disk during checkpointing anyway
Recovery: REDO log with checkpointing

**Log records**

- `<START T1>`
- `<T1, A, 5>`
- `<COMMIT T1>`
- `<T2, B, 10>`
- `<START CKPT(T2)>`
- `<T2, C, 15>`
- `<START T3>`
- `<T3, D, 20>`
- `<END CKPT>`
- `<COMMIT T2>`
- `<COMMIT T3>`

- `<COMMIT T2>` and `<COMMIT T3>` found after `<START CKPT>`:
  - both to be REDone
- REDO Update (in order)
  - `<T2, B, 10>`: B = 10
  - `<T2, C, 15>`: C = 15
  - `<T3, D, 20>`: D = 20
- Note: update has to be in the forward direction (redo log, unlike undo)

Recovery: REDO log with checkpointing

**Log records**

- `<START T1>`
- `<T1, A, 5>`
- `<COMMIT T1>`
- `<T2, B, 10>`
- `<START CKPT(T2)>`
- `<T2, C, 15>`
- `<START T3>`
- `<T3, D, 20>`
- `<END CKPT>`
- `<COMMIT T2>`
- `<COMMIT T3>`

- Example 1:
  - Crash before `<COMMIT T3>`
  - T3 has not committed
  - No need to redo for `<T3, D, 20>`

Recovery: REDO log with checkpointing

**Log records**

- `<START T1>`
- `<T1, A, 5>`
- `<COMMIT T1>`
- `<T2, B, 10>`
- `<START CKPT(T2)>`
- `<T2, C, 15>`
- `<START T3>`
- `<T3, D, 20>`
- `<END CKPT>`
- `<COMMIT T2>`
- `<COMMIT T3>`

- Example 2:
  - Crash before `<END CKPT>`
  - Need to find last `<END CKPT>` and then its `<START CKPT>`:
    - Here no other `<END CKPT>`
    - Scan until the start of the log
  - Only `<COMMIT T1>` found
    - Redo A = 5 for `<T1, A, 5>`

Pros and Cons

<table>
<thead>
<tr>
<th>UNDO</th>
<th>REDO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires data to be written to disk immediately after a transaction finishes — might increase the number of disk I/Os that need to be performed</td>
<td>Requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed — might increase the average number of buffers required by transactions</td>
</tr>
</tbody>
</table>

Also both may have conflicts during checkpointing with shared buffers:
- Suppose A in a page is changed by a committed transaction but B is changed by an uncommitted one
- Ok if no shared buffers

Get benefits of both — at the expense of maintaining more log records

UNDO/REDO logging

- `<T, X, v, w>`
  - T changed the value of element X
  - former value v
  - new value w
**UNDO/REDO logging rule**

(UR rule) Before modifying any element X on disk, <T, X, v, w> must appear on disk
- Only constraint imposed by both UNDO and REDO log
- no constraint on <COMMIT T>
- can precede or follow any of the changes to the db elements on disk

**UNDO: order of writing to disk**
1. <START T>
2. A = 8 (new value 8)
3. <COMMIT T>

**REDO: order of writing to disk**
1. <START T>
2. A = 12 (new value 12)
3. <COMMIT T>

**Recovery**

Recovery using UNDO/REDO logging

**REDO all committed transactions in the order earliest-first (forward)**

**UNDO all uncommitted/incomplete transactions in the order latest first (backward)**

**Important to do both**
- because of the flexibility allowed by UNDO/REDO logging regarding <COMMIT> records
- we can have
  - a committed transaction with not all changes written to disk
  - an uncommitted transactions with some changes written to disk

**EXAMPLE:**

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

**Crash example 1**
- Crash after <COMMIT T> is flushed to disk
- T is considered as committed
- First update A to 16
- Then update B to 16 (forward direction)
- Some changes may be unnecessary but harmless

**EXAMPLE: UNDO/REDO LOG**

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

**Crash example 2**
- Crash before <COMMIT T> is flushed to disk
- T is considered as uncommitted
- First update B to 8
- Then update A to 8 (backward direction)
- Some changes may be unnecessary but harmless
Checkpointing

Checkpointing process

1. Write log record \(<\text{START CKPT(T1, ..., Tk)}\rangle\) where \(T1, ..., Tk\) are the active (uncommitted) transactions, and flush the log

2. Write to disk all records that are dirty
   - i.e. contain one or more changed db elements
   - NOTE: unlike REDO logging, flush all dirty buffers – not just those written by committed transactions

3. Write a log record \(<\text{END CKPT}\rangle\) to the log and flush the log

A transaction must not write any value (even to memory buffers) until it is certain not to abort

An UNDO/REDO log with checkpointing

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

T2 is active

T2’s new B value will be written to disk when the checkpointing begins
- unlike UNDO, where first commit, then write

During CKPT,
- flush A to disk if it is not already there (dirty buffer)
- like REDO
- flush B to disk if it is not already there (dirty buffer)
- unlike REDO

Recovery:

An UNDO/REDO log with checkpointing

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

T1 has committed and writes on disk
- ignore T1
- REDO T2 and T3
- For T2
  - set C to 15
  - not necessary to set B to 10 (before END CKPT – already on disk)
- For T3
  - set D to 19
  - if T3 had started before START CKPT, would have had to look before START CKPT for more actions to be undone

Summary

- UNDO logging
  - \(<T, X, u>\) is the old value of X
  - \(<T, X, v>\) to disk \(\rightarrow\) X new value to disk \(\rightarrow\) <COMMIT T> to disk
  - undo uncommitted transactions
- REDO logging
  - \(<T, X, v>\) is the new value of X
  - \(<T, X, u>\) to disk \(\rightarrow\) X new value to disk \(\rightarrow\) <COMMIT T> to disk
  - redo committed transactions
- UNDO/REDO logging
  - \(<T, X, u, v>\) is the old value of X and v is the new value of X
  - \(<T, X, u, v>\) to disk \(\rightarrow\) X new value to disk
  - No constraints on writing <COMMIT T> to disk
  - both: undo uncommitted and redo committed transactions
- Understand for each of these three
  - standard recovery
  - checkpointing, and
  - recovery with checkpointing