Introduction to Database Systems
CSE 444

Lectures 9-10
Transactions: recovery

Outline
• We are starting to look at DBMS internals
• Today and next time: transactions & recovery
  – Disks 13.2 [Old edition: 11.3]
  – Undo logging 17.2
  – Redo logging 17.3
  – Redo/undo 17.4

The Mechanics of Disk

Mechanical characteristics:
• Rotation speed (5400RPM)
• Number of platters (1-30)
• Number of tracks (<=10000)
• Number of bytes/track(10^5)

Unit of read or write:
disk block
Once in memory: page
Typically: 4k or 8k or 16k

RAID
Several disks that work in parallel
• Redundancy: use parity to recover from disk failure
• Speed: read from several disks at once

Various configurations (called levels):
• RAID 1 = mirror
• RAID 4 = n disks + 1 parity disk
• RAID 5 = n+1 disks, assign parity blocks round robin
• RAID 6 = “Hamming codes”

Not required for exam, but interesting reading in the book

Disk Access Characteristics
• Disk latency = time between when command is issued and when data is in memory
• Disk latency = seek time + rotational latency
  – Seek time = time for the head to reach cylinder
  – 10ms – 40ms
  – Rotational latency = time for the sector to rotate
  – Rotation time = 10ms
  – Average latency = 10ms/2
• Transfer time = typically 40MB/s
• Disks read/write one block at a time

Large gap between disk I/O and memory ➔ Buffer pool

Design Question
• Consider the following query:

<table>
<thead>
<tr>
<th>SELECT</th>
<th>S1.temp, S2.pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>TempSensor S1, PressureSensor S2</td>
</tr>
<tr>
<td>WHERE</td>
<td>S1.location = S2.location AND S1.time = S2.time</td>
</tr>
</tbody>
</table>

• How can the DBMS execute this query given
  – 1 GB of memory
  – 100 GB TempSensor and 10 GB PressureSensor
Buffer Manager
- Enables higher layers of the DBMS to assume that needed data is in main memory
- Needs to decide on page replacement policy
  - LRU = expensive
  - Clock algorithm = cheaper alternative
- Both work well in OS, but not always in DB

Least Recently Used (LRU)

Buffer Manager
- DBMSs build their own buffer manager and don’t rely on the OS. Why?
  - Reason 1: Correctness
    - DBMS needs fine grained control for transactions
    - Needs to force pages to disk for recovery purposes
  - Reason 2: Performance
    - DBMS may be able to anticipate access patterns
    - Hence, may also be able to perform prefetching
    - May select better page replacement policy

Transaction Management and the Buffer Manager
- Recovery: ‘log-file write-ahead’, then careful policy about which pages to force to disk
- Concurrency control: locks at the page level, multiversion concurrency control

Transaction Management
- Recovery from crashes: ACID
- Concurrency control: ACID
**Problem Illustration**

Client 1:
```
START TRANSACTION
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99
DELETE Product
WHERE price <= 0.99
COMMIT
```

What do we do now?

---

**Recovery**

From which events below can DBMS recover?

- Wrong data entry
- Disk failure
- Fire / earthquake / etc.
- Systems crashes
  - Software errors
  - Power failures

---

**Recovery**

<table>
<thead>
<tr>
<th>Type of Crash</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong data entry</td>
<td>Constraints and Data cleaning</td>
</tr>
<tr>
<td>Disk crashes</td>
<td>Redundancy: RAID, backup, replica</td>
</tr>
<tr>
<td>Fire or other major disaster</td>
<td>Redundancy: Replica far away</td>
</tr>
<tr>
<td>System failures</td>
<td>DATABASE RECOVERY</td>
</tr>
</tbody>
</table>

Most frequent

---

**System Failures**

- Each transaction has *internal state*
  - When system crashes, internal state is lost
    - Don’t know which parts executed and which didn’t
    - Need ability to *undo* and *redo*
  - Remedy: use a *log*
    - File that records every single action of each transaction

---

**Transactions**

- Assumption: db composed of *elements*
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)
- Assumption: each transaction reads/writes some elements

---

**Primitive Operations of Transactions**

- READ(X,t)
  - copy element X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to element X
- INPUT(X)
  - read element X to memory buffer
- OUTPUT(X)
  - write element X to disk
Example

START TRANSACTION
READ(A,i);
  t := t*2;
WRITE(A,i);
READ(B,i);
  t := t*2;
WRITE(B,i);
COMMIT;

Atomicity: BOTH A and B are multiplied by 2

### Transaction

<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>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A)</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i:=t*2</td>
<td></td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td></td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td></td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i:=t*2</td>
<td></td>
<td>32</td>
<td>32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B)</td>
<td></td>
<td>32</td>
<td>32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Buffer pool

<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>READ(A)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>i:=t*2</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>i:=t*2</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>WRITE(B)</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Disk

<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>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A)</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i:=t*2</td>
<td></td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td></td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td></td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i:=t*2</td>
<td></td>
<td>32</td>
<td>32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B)</td>
<td></td>
<td>32</td>
<td>32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Buffer Manager Policies

- **STEAL or NO-STEAL**
  - Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?

- **FORCE or NO-FORCE**
  - Should all updates of a transaction be forced to disk before the transaction commits?

  - Easiest for recovery: NO-STEAL/FORCE
  - Highest performance: STEAL/NO-FORCE
**Solution: Use a Log**

- Log = append-only file containing log records
- Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - **Redo** some transactions that did commit
  - **Undo** other transactions that did not commit
- Three kinds of logs: undo, redo, undo/redo

---

**Undo Logging**

Log records

- **<START T>**
  - Transaction T has begun
- **<COMMIT T>**
  - T has committed
- **<ABORT T>**
  - T has aborted
- **<T,X,v>**  -- Update record
  - T has updated element X, and its old value was v

---

<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>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
<td></td>
</tr>
<tr>
<td>READ(A,i)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A,i)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,b&gt;</td>
<td></td>
</tr>
<tr>
<td>INPUT(B)</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>8</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>&lt;T,B,b&gt;</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
<td></td>
</tr>
</tbody>
</table>

---

**After Crash**

- In the first example:
  - We **UNDO** both changes: A=8, B=8
  - The transaction is atomic, since none of its actions has been executed
- In the second example
  - We don’t **undo** anything
  - The transaction is atomic, since both it’s actions have been executed

---
Undo-Logging Rules

U1: If T modifies X, then \(<T,X,v>\) must be written to disk before OUTPUT(X)

U2: If T commits, then OUTPUT(X) must be written to disk before <COMMIT T>

• Hence: OUTPUTs are done early, before the transaction commits

Recovery with Undo Log

After system’s crash, run recovery manager

• Idea 1. Decide for each transaction T whether it is completed or not
  – <START T>….<COMMIT T>…. = yes
  – <START T>….<ABORT T>……. = yes
  – <START T>…………………… = no

• Idea 2. Undo all modifications by incomplete transactions

Recovery with Undo Log

Recovery manager:

• Read log from the end: cases:
  – <COMMIT T>: mark T as completed
  – <ABORT T>: mark T as completed
  – <T,X,v>: if T is not completed then write X=v to disk else ignore
  – <START T>: Ignore

• Note: all undo commands are idempotent
  – If we perform them a second time, no harm done
  – E.g. if there is a system crash during recovery, simply restart recovery from scratch
Recovery with Undo Log

When do we stop reading the log?
- We cannot stop until we reach the beginning of the log file
- This is impractical

Instead: use checkpointing

Checkpointing

Checkpoint the database periodically
- Stop accepting new transactions
- Wait until all current transactions complete
- Flush log to disk
- Write a <CKPT> log record, flush
- Resume transactions

Undo Recovery with Checkpointing

During recovery,
Can stop at first <CKPT>

Nonquiescent Checkpointing

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- Idea: nonquiescent checkpointing

Nonquiescent Checkpointing

- Write a <START CKPT(T1,...,Tk)> where T1,...,Tk are all active transactions.
  Flush log to disk
- Continue normal operation
- When all of T1,...,Tk have completed, write <END CKPT>. Flush log to disk

Q: why do we need <END CKPT>?
Implementing ROLLBACK

- Recall: a transaction can end in COMMIT or ROLLBACK
- Idea: use the undo-log to implement ROLLBACK
- How?
  - LSN = Log Sequence Number
  - Log entries for the same transaction are linked, using the LSN’s
  - Read log in reverse, using LSN pointers

Redo Logging

Log records
- \(<\text{START } T>\) = transaction \(T\) has begun
- \(<\text{COMMIT } T>\) = \(T\) has committed
- \(<\text{ABORT } T>\) = \(T\) has aborted
- \(<T,X,v>\) = \(T\) has updated element \(X\), and its new value is \(v\)

Redo-Logging Rules

R1: If \(T\) modifies \(X\), then both \(<T,X,v>\) and \(<\text{COMMIT } T>\) must be written to disk before OUTPUT\((X)\)

- Hence: OUTPUTs are done late

Recovery with Redo Log

After system's crash, run recovery manager
- Step 1. Decide for each transaction \(T\) whether it is completed or not
  - \(<\text{START } T>\)....\(<\text{COMMIT } T>\).... = yes
  - \(<\text{START } T>\)....\(<\text{ABORT } T>\)..... = yes
  - \(<\text{START } T>\).................... = no
- Step 2. Read log from the beginning, redo all updates of committed transactions
Recovery with Redo Log

\[\text{<START T1>}\]
\[\text{T1,X1,v1}>\]
\[\text{<START T2>}\]
\[\text{T2,X2,v2}>\]
\[\text{<START T3>}\]
\[\text{T1,X3,v3}>\]
\[\text{<COMMIT T2>}\]
\[\text{T3,X4,v4}>\]
\[\text{T1,X5,v5}>\]
\[\ldots\]

Nonquiescent Checkpointing

- Write a \text{<START CKPT(T1,...,Tk)>}
  where T1,...,Tk are all active transactions
- Flush to disk all blocks of committed transactions (dirty blocks), while continuing normal operation
- When all blocks have been written, write \text{<END CKPT>}

Redo Recovery with Nonquiescent Checkpointing

Step 1: look for The last \text{<END CKPT>}

All OUTPUTs of T1 are known to be on disk

Cannot use

Step 2: redo from the earliest start of T4, T5, T6 ignoring transactions committed earlier

Comparison Undo/Redo

- Undo logging:
  - OUTPUT must be done early
  - If \text{<COMMIT T>} is seen, T definitely has written all its data to disk (hence, don’t need to redo) – inefficient
- Redo logging
  - OUTPUT must be done late
  - If \text{<COMMIT T>} is not seen, T definitely has not written any of its data to disk (hence there is no dirty data on disk, no need to undo) – inflexible
- Would like more flexibility on when to OUTPUT: undo/redo logging (next)

Undo/Redo Logging

Log records, only one change
- \text{<T,X,u,v>} = T has updated element X, its old value was u, and its new value is v

Undo/Redo-Logging Rule

\text{UR1: If T modifies X, then \text{<T,X,u,v>} must be written to disk before OUTPUT(X)}

Note: we are free to OUTPUT early or late relative to \text{<COMMIT T>}
Recovery with Undo/Redo Log

After system's crash, run recovery manager
- Redo all committed transaction, top-down
- Undo all uncommitted transactions, bottom-up

Granularity of the Log
- Physical logging: element = physical page
- Logical logging: element = data record
- What are the pros and cons?

Granularity of the Log
- Modern DBMS:
  - Physical logging for the REDO part
    - Efficiency
  - Logical logging for the UNDO part
    - For ROLLBACKs

Can OUTPUT whenever we want: before/after COMMIT